**Politecnico di Torino**

Master's Degree in Computer Engineering

A.y. 2023/2024

Graduation Session October 2024

# GUI Representation Learning for Downstream Real-World Applications

Supervisors:

Luigi De Russis

Tommaso Calò

Candidate:

Francesca Russo

**Abstract**

Recent advancements in Artificial Intelligence (AI) have allowed the development of tools that can assist professionals across various industries in completing different tasks, such as realistic character design in gaming, news analysis and fact-checking in journalism. Among the industries benefiting from AI advancements is User Interface (UI) design, where AI-based tools are playing a key role in enhancing efficiency and creativity.

Graphical User Interface (GUI) design is the process of designing the visual layout of a software application, focusing on the appearance, functionality and usability of the interface that users interact with. Nowadays, designers rely on several products, such as Figma, that facilitate the creation, prototyping and testing of GUIs. Figma, a real-time collaborative platform, has recently integrated novel AI features, with many more to come.

Unfortunately, in many datasets, GUIs are represented in a verbose format that may not be properly structured for achieving optimal performance for AI models. Moreover, existing GUI datasets are not built for seamless integration of AI models within Figma's environment.

To address these issues, a learned approach is proposed to extract a meaningful representation of GUI information, alongside the introduction of a new Figma-compatible hierarchical dataset. The objective is to facilitate both the development and the deployment of new AI models for downstream real-world applications in GUI design.

Specifically, this work involves training a Vector Quantized Variational Autoencoder (VQ-VAE) to learn a codebook of latent quantized embeddings, updated via Exponential Moving Average (EMA), capturing the relationships between GUI elements. This learned vocabulary can subsequently be used to train models for downstream real-world applications, such as GUI components generation. To validate the approach, the VQ-VAE is first trained and tested for bounding box reconstruction and category classification only, using both the well-known Rico dataset and the newly proposed Figma Layout User Interface Dataset (FLUID), which contains JSON files that can be directly imported in Figma. Next, GUI data from FLUID, incorporating additional elements such as image embeddings, text embeddings, and background colors, are integrated into the VQ-VAE to evaluate its ability to encode these more complex features.

The results demonstrate the VQ-VAE's capability to reconstruct and classify GUI items when trained on a limited set of features, particularly when using Rico. However, when more complex features are introduced using FLUID, the model

exhibits reduced performance, highlighting the need for further optimizations in both its architecture and training procedure.

# Acknowledgements

This journey has been full of invaluable experiences that have enriched my personal growth and academic knowledge. I would like to express my heartfelt gratitude to everyone who has participated, providing guidance, support and encouragement.

I would like to thank my supervisors, Professor Luigi De Russis and Tommaso Calò, for their invaluable guidance and insightful advice. Their expertise has been fundamental in shaping this work, and I am deeply grateful for their mentorship during this master thesis.

A special thank you is for my family. Words cannot fully express the gratitude I feel for allowing me to pursue this goal and for the unwavering support they have given me along the road.

To Claudio, my partner. Sharing this experience from nearly the first days to the very last day has been an incredible rollercoaster. Your emotional support, encouragement, and — above all — your belief in me have been the driving force behind my determination to give my best.

To everyone who has crossed my path and enriched this journey, even in the smallest way, thank you.

Completing this journey has been a challenging yet fulfilling adventure, and I am now ready to begin the next chapter of my life.

# Table of Contents

**List of Tables** VI

**List of Figures** VII

**1 Introduction** 1
- 1.1 Motivation and Objective . . . 2
- 1.2 Structure of the Thesis . . . 3

**2 Background** 4
- 2.1 Introduction to GUI Design . . . 4
  - 2.1.1 Design Principles and Guidelines . . . 5
  - 2.1.2 The User Interface Design Process . . . 6
- 2.2 Machine Learning for GUI Design . . . 7
  - 2.2.1 The Role of Machine Learning for GUI Design . . . 7
  - 2.2.2 GUI Representation Learning . . . 8
  - 2.2.3 Rico Dataset . . . 10
- 2.3 Autoencoders and their Variants . . . 11

**3 FLUID Dataset** 15
- 3.1 Related GUI Datasets . . . 15
  - 3.1.1 Klarna Dataset . . . 15
  - 3.1.2 WebColor Dataset . . . 16
- 3.2 Filling the Gaps in GUI Datasets with FLUID . . . 17
- 3.3 Data Collection . . . 18
- 3.4 Data Analysis and Cleaning . . . 26
- 3.5 Conclusion . . . 34

**4 Methodology** 36
- 4.1 Bounding Box and Category Encoded Representation Learning . . . 36
  - 4.1.1 Model Architecture . . . 37
  - 4.1.2 Learning Procedure . . . 41

4.1.3 Implementation Details . . . . . . . . . . . . . . . . . . . . . . 42
4.2 Multimodal Encoded Representation Learning . . . . . . . . . . . . 45
4.2.1 Multimodal Input Representation . . . . . . . . . . . . . . . . 45
4.2.2 Multimodal Learning Procedure . . . . . . . . . . . . . . . . . 47
4.2.3 Implementation Details . . . . . . . . . . . . . . . . . . . . . . 48

**5 Results** 50
5.1 Bounding Box and Category Encoded Representation Learning . . . 50
5.1.1 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . 50
5.1.2 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . 52
5.2 Multimodal Encoded Representation Learning . . . . . . . . . . . . 61
5.2.1 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2.2 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . 62

**6 Conclusion** 74

**Bibliography** 76

# List of Tables

3.1 Description of GUI elements' attributes in FLUID JSON files . . . . 21
3.2 Description of GUI nodes' categories in FLUID JSON files . . . . . 22

5.1 VQ-VAE evaluation results for bounding box reconstruction and category prediction on Rico and FLUID . . . . . . . . . . . . . . . . 52
5.2 VQ-VAE mIoU evaluation results for bounding box reconstruction on Rico and FLUID, including the test without small GUI elements 60
5.3 VQ-VAE evaluation results on FLUID for multimodal representation learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.4 Examples of text correctly reconstructed by the VQ-VAE512 and VQ-VAE768 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.5 Examples of text wrongly reconstructed by the VQ-VAE512 and VQ-VAE768 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

# List of Figures


2.1 GUI of the Amazon Shopping mobile application . . . . . . . . . . 5
2.2 Top navigation bar in Trello . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Four GUIs randomly selected from the Rico dataset . . . . . . . . . 11
2.4 Variational Autoencoder model architecture . . . . . . . . . . . . . 12
2.5 Vector Quantized-Variational Autoencoder model architecture . . . 13

3.1 GUI samples from the WebColor training set . . . . . . . . . . . . . 17
3.2 Visual aspect of a WebColor web page with and without the cookie notification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.3 Figma plugin Builder.io-Figma to HTML, React and more interface 24
3.4 Visual comparison between two WebColor web pages and their corresponding FLUID GUIs imported in Figma . . . . . . . . . . . 25
3.5 Distribution of the total number of elements per file in FLUID . . . 26
3.6 Distribution of the number of images per file in FLUID . . . . . . . 27
3.7 Distribution of the number of SVGs per file in FLUID . . . . . . . . 27
3.8 Distribution of the number of textual elements per file in FLUID . . 28
3.9 Distribution of the length of textual elements, in number of words, in FLUID . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.10 Distribution of the attributes x, y, width and height in FLUID . . . 29
3.11 Example of a FLUID GUI opened in Figma, where width is greater than 375 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.12 Distribution of GUI elements' attributes in FLUID dataset . . . . . 31
3.13 Effect of removing the attributes topLeftRadius, topRightRadius, bottomLeftRadius, and bottomRightRadius from FLUID GUIs . . . 32
3.14 Example of a grey box negatively impacting the visual layout of a FLUID GUI opened in Figma . . . . . . . . . . . . . . . . . . . . . 33
3.15 Distribution of the total number of elements per file in FLUID after data cleaning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.16 Operations performed on the WebColor dataset to build FLUID . . 35

4.1 Proposed VQ-VAE model architecture . . . . . . . . . . . . . . . . . 37

4.2 Proposed VQ-VAE encoder module architecture . . . . . . . . . . . 38
4.3 Proposed VQ-VAE Vector Quantizer module . . . . . . . . . . . . . 39
4.4 Proposed VQ-VAE decoder module architecture . . . . . . . . . . . 40

5.1 Average area and position errors for Rico . . . . . . . . . . . . . . . 54
5.2 Average area and position errors for FLUID . . . . . . . . . . . . . 54
5.3 Example #1 of GUI from FLUID and its VQ-VAE reconstruction . 55
5.4 Example #2 of GUI from FLUID and its VQ-VAE reconstruction . 55
5.5 Example #3 of GUI from FLUID and its VQ-VAE reconstruction . 56
5.6 Example #4 of GUI from FLUID and its VQ-VAE reconstruction . 56
5.7 Example #1 of GUI from Rico and its VQ-VAE reconstruction . . . 57
5.8 Example #2 of GUI from Rico and its VQ-VAE reconstruction . . . 58
5.9 Example #3 of GUI from Rico and its VQ-VAE reconstruction . . . 58
5.10 Example #4 of GUI from Rico and its VQ-VAE reconstruction . . . 59
5.11 Distribution of the areas for FLUID and Rico bounding boxes of the test set . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.12 Distribution of the areas for FLUID and Rico bounding boxes of the test set, on a restricted interval . . . . . . . . . . . . . . . . . . . . 60
5.13 VQ-VAE512 average area and position errors for FLUID . . . . . . 63
5.14 VQ-VAE768 average area and position errors for FLUID . . . . . . 63
5.15 Example #1 of GUI from FLUID and its VQ-VAE512 reconstruction 64
5.16 Example #2 of GUI from FLUID and its VQ-VAE512 reconstruction 65
5.17 Example #3 of GUI from FLUID and its VQ-VAE512 reconstruction 65
5.18 Example #4 of GUI from FLUID and its VQ-VAE512 reconstruction 66
5.19 Example #1 of GUI from FLUID and its VQ-VAE768 reconstruction 67
5.20 Example #2 of GUI from FLUID and its VQ-VAE768 reconstruction 67
5.21 Example #3 of GUI from FLUID and its VQ-VAE768 reconstruction 68
5.22 Example #4 of GUI from FLUID and its VQ-VAE768 reconstruction 68
5.23 Examples of images correctly reconstructed by the VQ-VAE512 . . 69
5.24 Examples of images wrongly reconstructed by the VQ-VAE512 . . . 69
5.25 Examples of images correctly reconstructed by the VQ-VAE768 . . 70
5.26 Examples of images wrongly reconstructed by the VQ-VAE768 . . . 70
5.27 Examples of SVGs correctly reconstructed by the VQ-VAE512 . . . 71
5.28 Examples of SVGs wrongly reconstructed by the VQ-VAE512 . . . 71
5.29 Examples of SVGs correctly reconstructed by the VQ-VAE768 . . . 72
5.30 Examples of SVGs wrongly reconstructed by the VQ-VAE768 . . . 72

# Chapter 1

# Introduction

In the last two decades, the digital landscape has been enriched by the continuous release of applications designed to meet nearly every conceivable need in daily life. From managing personal finances and staying connected with loved ones to ordering food and navigating cities, there is an app for virtually everything. As reliance on these applications grows, the design and functionality of their Graphical User Interfaces (GUIs) have become critical factors in determining user satisfaction and engagement, as these serve as the medium of interaction between users and the application itself.

The creation of an effective GUI is a complex and frequently time-consuming process that requires careful consideration of several factors. Designers must balance aesthetics with functionality, ensuring that interfaces are not only visually appealing but also intuitive and user-friendly. The design must be human-centered, allowing all users, regardless of their abilities, to access the application, and it should be tailored to accommodate the diverse needs and behaviors of its user base.

Moreover, the process of designing GUIs extends beyond just the visual aspect; it involves understanding the psychological and cognitive aspects of how users interact with digital environments. The role of designers is to anticipate user needs, predict potential interactions, and create a seamless experience that smoothly guides users through tasks. This requires a deep understanding of user experience (UX) principles and often involves iterative testing and refinement to achieve the desired outcomes. Given the importance of these aspects, it becomes clear that the design of GUIs is not simply a creative process but a critical component of application development that directly impacts user engagement and satisfaction.

As a consequence, designers often rely on several tools that facilitate the creation, prototyping and testing of GUIs. The use of these tools significantly enhances the efficiency and effectiveness of the GUI design process, ensuring that the resulting interfaces are well-crafted and ready for implementation. Among these tools, Figma [1] is one of the most widely used in the industry. It provides a real-time

collaborative environment that supports both the aesthetic and functional aspects of design. Therefore, Figma plays a crucial role in helping designers navigate the complexities of modern GUI development, ultimately contributing to the creation of applications that satisfy the users' expectations.

Recent advancements in Artificial Intelligence (AI) have led to the development of intelligent assistants that can help professionals across various industries, including User Interface (UI) design. By automating repetitive tasks, providing design recommendations, and even predicting user preferences through data analysis, AI has the potential to considerably improve the efficiency and creativity of the GUI design process. For example, AI-powered systems can be trained on extensive datasets of user interactions to identify patterns and provide suggestions on how to optimize the user interface according to common user behaviors. Figma, for instance, has recently introduced new AI-powered features, with many more updates expected to follow soon.

As AI continues to evolve, its role in GUI design is likely to expand, offering even more sophisticated tools that can assist designers in producing interfaces tailored to the needs and preferences of the users.

## 1.1 Motivation and Objective

Many datasets, such as the Klarna Product Page Dataset [2], present GUIs in an HTML-like format that is often not structured in a way that optimizes performance for a wide range of AI-driven real-world applications. Training AI models directly on raw GUI data can result in suboptimal learning dynamics, where models struggle to focus on relevant features due to the presence of redundant information or data noise. This interferes with the models' ability to effectively generalize and perform across diverse tasks.

Moreover, these HTML-like data formats are not compatible with real design frameworks, such as Figma. Consequently, AI models trained on these datasets are not directly deployable in a real design environment.

Given these issues, the objective of this thesis is to facilitate both the development and the deployment of new AI models to be used in real design frameworks, to support and enhance the work of designers.

To achieve this, this work proposes:

- **A learned encoded representation of GUIs** that can be used for downstream tasks, including discrimination, palette and typography recommendation, and generation of new components. Starting from raw GUI data, a Deep Learning (DL) model is used to extract the most important features, that can later be exploited for training downstream AI models. In this way, these can be efficiently trained on a meaningful and refined set of features. Moreover,

the encoded representation is also suitable for GUI elements tokenization to be used in a cross-modal learning framework, where a model is trained on information from multiple modalities (e.g., text, images).

- **A new Figma-compatible dataset** composed of real-world GUIs derived from the WebColor dataset [3]. The original data are manipulated by means of a Figma plugin to produce an alternative hierarchical data representation of GUI components suitable for a variety of AI-driven tasks. The compatibility with Figma allows downstream applications, built upon this dataset, to be seamlessly integrated in the Figma environment, ultimately assisting and enhancing the design experience. In fact, designers are often not familiar with HTML-based GUI representations and can greatly benefit from the introduction of new tools in design frameworks they're already accustomed to.

## 1.2 Structure of the Thesis

The thesis is organized as follows:

- Chapter 2 provides an introduction to the GUI design process and how Machine Learning (ML) fits into this context.
- Chapter 3 describes the steps performed to obtain a new and well-curated dataset of GUIs, specifically built for compatibility with Figma.
- Chapter 4 outlines the model architecture, the learning procedure and the implementation details of the proposed AI models for representation learning.
- Chapter 5 reports the metrics used to evaluate the proposed AI models, and the results of these models.
- Finally, Chapter 6 summarizes the findings of this research, highlighting the potential strengths and limitations for AI-based representation learning.

# Chapter 2

# Background

This chapter provides the necessary background information to understand the Graphical User Interface (GUI) Design process and how Machine Learning (ML) fits into this context, also highlighting specific GUI datasets and model architectures often leveraged for ML development in the GUI design scenario.

Specifically, Section 2.1 serves as an introduction to the traditional GUI design process for those who may not be familiar with the topic.

Then, Section 2.2 first explains the role of machine learning for GUI design, highlighting the benefits of this joint collaboration in light of the previous overview of the traditional GUI design process. Subsequently, the most relevant works for GUI representation learning are presented, along with an introduction to the Rico [4] dataset, often used for the development of ML algorithms for GUI design.

Finally, Section 2.3 presents a general introduction to Autoencoders (AEs) and their variants, which are particularly relevant for GUI representation learning.

## 2.1 Introduction to GUI Design

**GUI design** is a discipline that deals with the design and development of the graphical interface of a software product.

GUI design focuses on maximizing the usability and the user experience, as the GUI is the point of contact between the user and the underlying functionalities of the software. Effective GUI design ensures that user interactions are intuitive, efficient and aesthetically pleasing, thereby facilitating seamless navigation and engagement between the users and the software.

### 2.1.1 Design Principles and Guidelines

Over the years, several **principles** and **guidelines** have been proposed to help designers produce GUIs that meet high quality design standards. The principles are general concepts that guide the overall approach to GUI design, while the guidelines are practical actionable recommendations, often derived from principles, aimed at solving specific design problems.

One notable example is **Nielsen's set of 10 usability heuristics** [5], which provides widely recognized principles for enhancing user interface design.

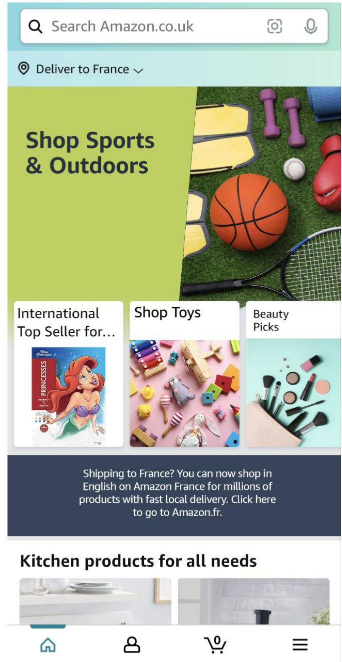



**Figure 2.1:** GUI of the Amazon Shopping mobile application [6]

Figure 2.1 shows the GUI of the Amazon Shopping mobile application. The magnifying glass icon is universally recognized as the icon for a search functionality. Nielsen's fourth principle states:

> *"Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform and industry conventions."*

Therefore, it would not make sense to use a different icon for the search functionality, as it would only force users to learn something new and make the overall experience odd. The Amazon Shopping GUI adheres to the principle and, as a consequence, is expected to deliver a positive user experience to customers.

GUI designers also exploit perception principles, called **Gestalt principles** [7], that are based on how the human brain perceives and recognizes groups of elements.

The goal is to create a hierarchical structure for GUI elements to guide the user to understand where the information is located, what are the most important elements in the application and how different elements of the user interface are related to each other. This way, the users are able to create a mental map of the application and can easily interact with the GUI to achieve their goals.

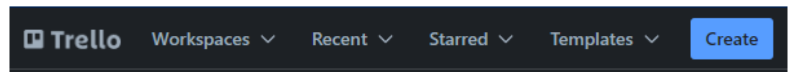



**Figure 2.2:** Top navigation bar in Trello [8]

The top navigation bar in Trello, shown in Figure 2.2, takes advantage of the fact that objects that are spatially close appear linked to each other. The buttons' placement in the bar creates a visual hierarchy that strengthens the user's mental map of the system: the buttons not only are similar in aspect, they are adjacent too. This is a clear example of the Gestalt *proximity* principle.

Overall, the design of a GUI is a complex process that requires balancing aesthetics and usability. The final output should be a user interface both practical and visually appealing, with creativity that also plays an important role in finding ways to captivate the user's attention and keep him engaged.

### 2.1.2 The User Interface Design Process

The process of designing a GUI involves a structured user-centered approach that integrates the previously mentioned principles and guidelines.

The whole GUI design process can be summarized in the following steps:

1. **Collection of all the necessary requirements for the project**. This involves interacting with potential users to understand who they are, what their needs are and what goals they want to achieve using the application under development.

2. **Identification of the "paths" expected to be taken by the users within the interface**. This requires to define all the high-level steps users will go through to complete key tasks, ensuring the process is logical and efficient.

3. **Production of a simple, low-detail set of sketches of the interface layout**. These sketches, often referred to as *wireframes*, serve as a blueprint for the interface's structure, initially focusing on functionality rather than visual design, and are often iteratively refined.

4. **Design of the visual aspect of the interface**. This includes selecting colors, typography, and designing buttons, icons, and other User Interface (UI) elements to create a polished look. A mid-fidelity prototype is produced and possibly tested with potential users to further refine it.

5. **Development of an interactive version of the design to allow users to experience the flow and the functionalities offered by the application**. In this context, usability testing with real users can be conducted to gather feedback on how well the design works and identify issues, if any.

6. **Application of the necessary adjustments to the design**, to better target user needs, based on the feedback from user testing.

Throughout these steps, designers often leverage specialized tools to streamline the design process and enhance the quality of their work. These tools not only help in visualizing the structure and flow of the interface but also facilitate collaboration among team members. By integrating such tools into the design process, designers can efficiently iterate on their prototypes and ensure that the final product aligns with users' expectations and project's goals. One very famous and powerful tool is Figma [1]. It allows designers to create, prototype, and also collaborate on user interface designs in real-time.

Starting from the fifth step, the development phase, and in the successive refinements, designers work closely with developers to ensure that the design is implemented as intended. This collaboration involves providing detailed design specifications and being available to answer questions and solve any issues that may arise during development.

Once the application is built, it will be made available to the users. After the official launch, the application is continuously monitored to understand how users interact with it and to gather valuable data on its performance. These data are used for future improvements.

## 2.2 Machine Learning for GUI Design

### 2.2.1 The Role of Machine Learning for GUI Design

Given the description of the GUI design process in Section 2.1.2, it is evident that **it is a complex and time-consuming task**. Moreover, **it requires proper collaboration in a cross-functional team to achieve the desired outcome**.

It is clear that Artificial Intelligence (AI) tools can be particularly useful to streamline the different design phases, providing a certain level of *assistance* that depends on the specific task these tools are meant to address.

For instance, the second step of the GUI design process (i.e., identification of the "paths" expected to be taken by the users within the interface) could be partially automated by training user agents in a reinforcement learning framework to complete some tasks in GUIs while mimicking human behavior [9]. The user agent can be used to assist the designer, uncovering patterns that may have been missed or producing a list of "paths" and related tasks the designer may decide to start from, rather than starting from scratch.

For the fourth step of the GUI design process (i.e., design of the visual aspect of the interface), one of the many possibilities is to develop an AI model able to assign colors to UI elements [3]. This task is known as *colorization*, and the goal is to provide a tool to assist designers in the transition from a low-detail sketch to a mid-fidelity prototype.

At this stage, the prototype is complex enough to leverage its visual and functional context, and provide a broad range of intelligent assistants. Figma [1], for example, now offers new AI-powered features able to understand the content of a static mock and automatically turn it into an interactive prototype, all with the touch of a button.

As research in AI continues to grow, new, more accurate and sophisticated tools are expected to improve the designer's workflow, while also accounting for the user needs and preferences.

### 2.2.2 GUI Representation Learning

In the past few years, significant advancements have been made also in the development of learned representations of GUIs. These learned representations are able to encode the essential visual characteristics of the interfaces and could prove to be useful for a variety of downstream tasks such as GUI generation, recommendation of designs and more. These AI models for representation learning can improve the efficacy of model training, enabling better generalization across a variety of GUI design tasks by simplifying complex interfaces into compact and meaningful representations [4, 10, 11, 12].

Several approaches leverage Transformer [13] architectures to capture dependencies within GUIs. Arroyo et al. [14] present a novel model that generates realistic and diverse layouts by integrating Transformer Networks and Variational Autoencoders (VAEs) [15]. The model learns a latent representation of GUI layouts and captures dependencies between layout components in an unsupervised manner by leveraging transformer-based attention mechanisms. This allows designers to explore innovative layouts by sampling from the learned representation. Similarly, Sobolevsky et al. [16] introduce GUILGET, a technique that transforms GUI elements into discrete representations. A token is assigned to each GUI component based on its semantic type and spatial information, such as its position and size

within the layout. Subsequently, the transformer model processes these tokens, which represent components and their relationships. The transformer ensures that the generated layout complies with design constraints (e.g., alignment and containment) by capturing dependencies between tokens. This is achieved by employing attention mechanisms and minimizing losses that are responsible for placing each component within its parent layout, preventing component overlap if they are within the same parent, and enabling component alignment.

Yamaguchi et al. [17] introduce CanvasVAE, which learns a structured representation of vector graphics leveraging on a variational autoencoder. The model encodes vector graphic commands into latent representations, capturing the relationships between graphical elements like shapes, paths, and text.

Lee et al. [18] introduce a method for generating graphic layouts that is based on user-specified constraints. This method employs Graph Neural Networks (GNNs) [19] to encode the relationships between design components (e.g., images, text) as nodes and edges in a graph.

Generative Adversarial Networks (GANs) [20] are used by Kikuchi et al. [21] for constraint-based layout generation. In their work, they optimize the latent space of a pre-trained GAN to ensure the layout meets the constraints without requiring retraining. The system generates high-quality layouts that satisfy user-specified rules by using latent space exploration and optimization.

Screen2Vec [10] generates semantic embeddings for GUI components and GUI display screens. The text label and class embeddings for the element's category are encoded using Sentence-BERT [22]. Subsequently, the component-level embedding is generated by combining these embeddings through a linear layer. The component embeddings are combined at the screen level using a Recurrent Neural Network (RNN) [23], which is then supplemented by layout embeddings using an autoencoder and app metadata embeddings using Sentence-BERT. A final linear layer is employed to integrate all three embeddings to obtain the screen-level embedding.

Creating an encoded representation plays an essential role in UI retrieval tasks, facilitating the extraction and representation of both visual and structural characteristics of UIs [12, 11]. Wu et al. [12] introduce a method to parse GUIs from screenshots using a Convolutional Neural Network (CNN) [24]. Their approach extracts pixel-level features that are, subsequently, used to predict a structured representation of the GUI, producing an encoding that captures both the visual and structural relationships between UI components. This procedure efficiently converts screenshots into a format that can be employed for subsequent tasks, such as UI retrieval, similarity matching, or reverse engineering UI code. Similarly, Huang et al. [11] use two convolutional sub-networks (i.e., one for sketches, one for screenshots) based on the VGG-A [25] architecture, each of them outputting an embedding. Each sub-network is trained with a triplet loss function to guarantee that the embeddings of sketches and their respective UIs are similar. In this manner,

the model learns a shared embedding space, which facilitates the retrieval of the correspondent UI based on sketches.

Some approaches try to combine the textual content and the visual design to enhance GUI representation [26, 27]. Li et al. [27] explore how to translate natural language instructions into executable sequences of actions on mobile UIs. It employs a transformer model to extract key action phrases from instructions and map them to UI objects based on content and screen position. Pasupat et al. [26] employ retrieval techniques combined with learned embeddings of web elements and commands to perform spatial, relational, and functional reasoning, linking commands to corresponding web elements effectively.

Semantic annotation of GUIs is addressed by Liu et al. [28], who create a lexical database that contains a set of design semantics to automatically annotate mobile GUIs. This approach has been demostrated to be capable of identifying the category of a GUI component, the semantics of buttons, and the overall task flow of screens. They employ an autoencoder to generate a low-dimensional learned representation, which is then used to conduct a nearest neighbor search.

Finally, Jing et al. [29] propose LayoutVQ-VAE, a model capable of generating layouts with internal and external constraints while relying on a discrete latent representation for the layout.

### 2.2.3 Rico Dataset

One of the most used datasets in AI research applied to GUI design is the Rico [4] dataset.

It has been widely used in various AI research areas, including UI design automation and generation [30, 31, 18, 21, 32, 14, 16, 17], usability prediction [33], GUI layout synthesis [34, 35], UIs retrieval [12, 11], and visual embedding learning [10, 28], demonstrating its versatility in supporting the development of advanced machine learning models. Some of these works have also been briefly introduced in Section 2.2.2.

Specifically, Rico is a large dataset of mobile app UIs aimed at supporting research in UI design and Human-Computer Interaction (HCI). Introduced in 2017, it was created to capture a wide variety of design patterns and user flows from Android applications. The dataset contains over 72,000 unique UI screens across more than 9,700 mobile apps spanning 27 categories from the Google Play Store, making it one of the most comprehensive resources for understanding mobile app design.

Each entry in the Rico dataset includes not only the visual representation of the UI screen, but also metadata about its structure, such as view hierarchies and UI components. This rich combination of data representations allows researchers to analyze the relationships between different UI elements and how they contribute

to the overall user experience. For instance, elements like buttons, text fields, and images are labeled, for a **total number of GUI element categories of 25**, and their properties, including layout positions and attributes, are available. This makes the dataset particularly valuable for machine learning tasks involving the analysis of GUI layouts.

Some examples of GUIs extracted from the Rico dataset are shown in Figure 2.3.

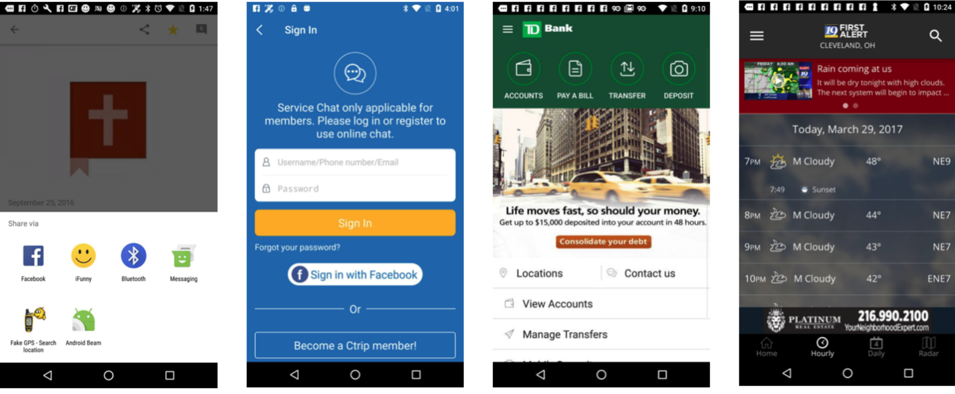



**Figure 2.3:** Four GUIs randomly selected from the Rico dataset [4]

## 2.3 Autoencoders and their Variants

Autoencoders (AEs) are neural networks specifically designed for learning an alternative representation of the input data in an unsupervised manner. Given their relevancy and the frequent use of their variants (e.g., VAEs, VQ-VAEs) for GUI representation learning in the works reported in Section 2.2.2, a general introduction is provided below.

**Autoencoders** are composed of two parts:

- An **encoder** that compresses the input into a latent space representation.
- A **decoder** that reconstructs the original input starting from the previously generated latent representation.

Traditional autoencoders learn deterministic mappings, which limits their ability to generate new and diverse data samples.

**Variational Autoencoders** (VAEs) [15] were introduced to overcome this limitation by employing a probabilistic method for encoding. Instead of mapping inputs to fixed points in the latent space, VAEs learn a distribution and map the

inputs as points into a probability distribution, typically a normal distribution. The data distribution in the latent space is represented by its mean $\mu$, the center of the distribution in the latent space, and its variance $\sigma$, that controls the spread of the distribution.

Figure 2.4 shows the architecture of the VAE, which consists of two components: the **probabilistic encoder** and the **probabilistic decoder**.

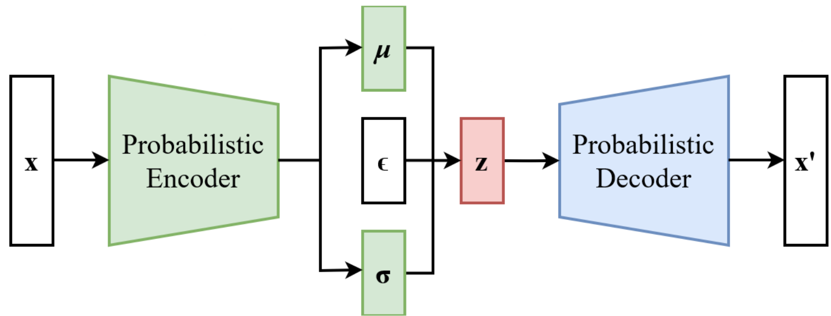



**Figure 2.4:** Variational Autoencoder model architecture [36]

The encoder maps the input data $x$ to the mean $\mu$ and the variance $\sigma$ of the data distribution. Sampling directly from the learned latent distribution would break the ability to backpropagate, as the sampling introduces non-differentiable randomness. Therefore the reparameterization trick is used. A random variable $\epsilon$ is sampled from a standard normal distribution (i.e., $\mu = 0$ and $\sigma = 1$) and is used to compute a latent vector $z$ defined as in Equation 2.1.

$$z = \mu(x) + \sigma(x)\epsilon \tag{2.1}$$

By learning a distribution rather than a single point for each input, the model gains flexibility. This allows the VAE to generate **new, similar data points** by sampling from the learned distributions.

The decoder takes the sampled latent vector $z$ and tries to reconstruct the original input $x$.

Therefore, **the objective of the VAE is to model the underlying data distribution in a way that captures meaningful latent variables**. To achieve this, it is necessary to compute the posterior distribution $p(z|x)$, which represents the probability of the latent variable $z$ given the observed data $x$.

The true posterior is intractable because it requires computing the exact likelihood of the data given the latent variable $z$. For this reason, VAEs aim to approximate the intractable posterior $p(z|x)$ with a simpler distribution $q_\phi(z|x)$ by optimizing the Evidence Lower Bound (ELBO) as in Equation 2.2.

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x) \| p(z)) \tag{2.2}$$

where:

- $p_\theta(x|z)$ is the likelihood of the data given the latent variable.
- $q_\phi(z|x)$ is the approximate posterior.
- $p(z)$ is the prior over the latent variable, usually a normal distribution.
- $D_{KL}$ is the Kullback-Leibler divergence measuring the difference between two probability distributions.

The first term is the *reconstruction loss*, that measures how well the decoder reconstructs the original input, while the second term is the *divergence loss* (DL) that encourages the learned latent space to be smooth and continuous.

VAEs have been utilized across various domains due to their ability to model complex data distributions and learn meaningful latent representations, such as image generation [37, 38], semi-supervised learning [39, 40], anomaly detection [41], data imputation [42, 43], 3D object modeling [44, 45], speech and audio processing [46, 47].

While VAEs use continuous latent spaces, **Vector Quantized-Variational Autoencoders** (VQ-VAEs) [48] replace the continuous latent variables with discrete *codebook* embeddings, combining the representational power of VAEs with the benefits of a discrete latent space.

Discrete latent spaces are more effective at capturing the categorical and symbolic information inherent in data, leading to better clustering and more interpretable latent spaces.

Figure 2.5 shows the overall VQ-VAE model architecture.

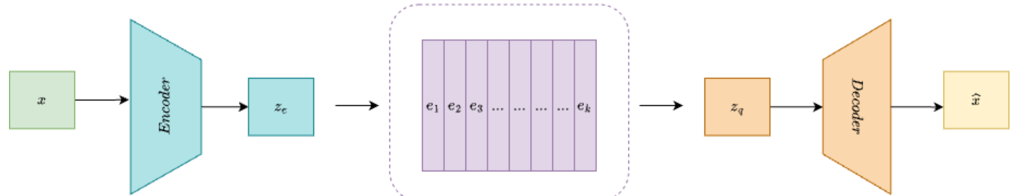



**Figure 2.5:** Vector Quantized-Variational Autoencoder model architecture

Starting from the input data $x$, the encoder outputs continuous vectors $z_e$ that are mapped to the nearest vector in the codebook of embeddings $e_i$, as reported in Equation 2.3.

$$z_q = e_k \text{ where } k = \arg\min_i ||z_e - e_i|| \tag{2.3}$$

The decoder then tries to build the reconstruction $\hat{x}$ of the original input $x$ starting from the discrete vector $z_q$.

The loss function for training a VQ-VAE is usually a combination of a reconstruction term $\mathcal{L}_{reconstruction}$ and a commitment term $\mathcal{L}_{commitment}$, as pointed out in Equation 2.4.

$$\mathcal{L} = \mathcal{L}_{reconstruction} + \mathcal{L}_{commitment} \tag{2.4}$$

where:

- $\mathcal{L}_{reconstruction}$, defined as $||x - \hat{x}||_2^2$, encourages the decoder to produce accurate reconstructions.
- $\mathcal{L}_{commitment}$ is defined as $||z_e - sg(e_k)||_2^2$, where $sg(.)$ is the stop gradient operator. This loss encourages the encoder to produce continuous representations as close as possible to the vectors of the codebook.

VQ-VAEs are widely used to model complex data distributions by employing discrete latent spaces, which better capture categorical information, making them ideal for a variety of tasks such as high-quality image generation [49, 50], audio synthesis and speech generation [51, 52], unsupervised learning [53], compressing images and videos [54, 55].

# Chapter 3

# FLUID Dataset

Following the discussion in Section 1.1, this chapter presents **FLUID**, Figma Layout User Interface Dataset, a well-curated collection of Graphical User Interfaces (GUIs), derived from the WebColor [3] dataset and specifically adapted for compatibility with Figma [1]. This chapter outlines the motivations and the steps taken to collect and refine the dataset, transforming HTML-based GUI representations into Figma-compatible JSON files.

Specifically, Section 3.1 introduces the main features of the Klarna Product Page Dataset [2], from which the WebColor dataset [3] is derived from, and the most relevant information about WebColor itself.

Then, Section 3.2 contextualizes the two datasets in a joint Artificial Intelligence (AI)-GUI design framework, highlighting their current limitations and setting the stage for the introduction of FLUID.

So, Section 3.3 reports the operations performed on the WebColor dataset to create FLUID, along with descriptive information about the new proposed dataset.

After this, FLUID is analyzed and further refined as described in Section 3.4.

At the end, the overall procedure to create FLUID is summarized in Section 3.5 to provide a concise reference that can be easily consulted.

## 3.1 Related GUI Datasets

### 3.1.1 Klarna Dataset

The **Klarna Product Page Dataset** [2] includes publicly available shopping web pages. It contains 51,701 product pages collected from 8,175 retailers in 8 distinct countries (US, GB, SE, NL, FI, NO, DE, AT), between 2018 and 2019.

The dataset was originally created to facilitate the development of AI systems for predicting web GUI elements.

All Klarna web pages include only mobile-sized GUIs in portrait mode. Each sample is available in three data formats:

- **MIME HTML (MHTML) files**: web archive files that capture the entire HTML content along with embedded resources like images and scripts.
- **WebTraversalLibrary (WTL) snapshots**: structured data formats created by the WebTraversalLibrary [56], a tool designed for scraping and analyzing web content. WTL snapshots contain structured data about the web page's DOM (Document Object Model).
- **Screenshots**: image files that capture the visual rendering of the page.

These data formats could be used in different ways to build a new GUI dataset, allowing for flexibility in choosing how to manipulate the source data to reach the desired outcome. Therefore, the Klarna Dataset is an interesting candidate for assembling the new dataset.

### 3.1.2 WebColor Dataset

**WebColor** [3] is a dataset obtained by filtering and polishing GUIs from the Klarna Product Page Dataset. The applied modifications are intended to produce a dataset suitable for the web page colorization task.

The operations performed on the Klarna Dataset to create WebColor are:

1. Filtering elements that do not appear on the screen, including alternative elements that only appear on laptop sized screens and functional elements, such as hidden input elements in submission forms, effectively reducing redundancy in the HTML of mobile GUIs.
2. Addition of a progressive selector to HTML elements (e.g., `#element-1`).
3. Retrieval of color values computed by the browser and simplification of web pages using both the Selenium WebDriver [57] and Google Chrome [58].
4. Removal of web pages that, even after simplification, still have more than 200 elements.
5. Exclusion of web pages that are encoded differently from UTF-8.

After these operations are carried out, the resulting WebColor dataset contains 44,048 product pages divided as:

- 27,630 training samples.

- 3,190 validation samples.
- 13,228 test samples.

Some examples of GUIs extracted from the WebColor dataset are shown in Figure 3.1.

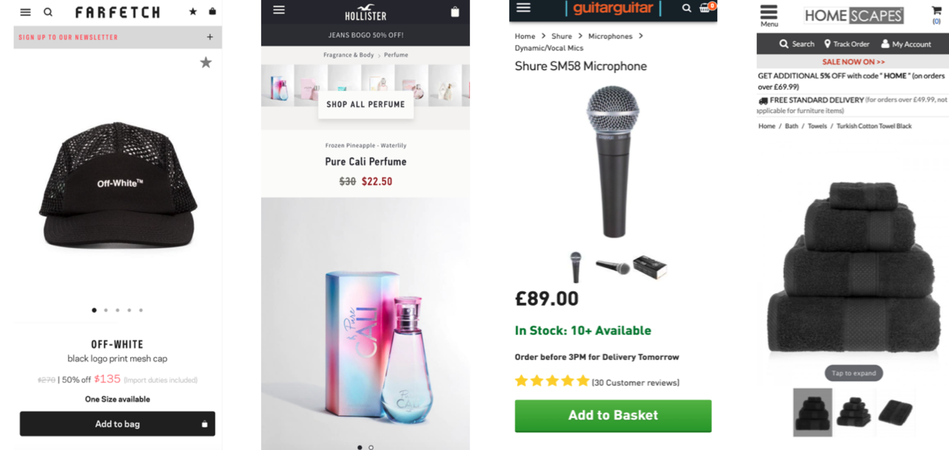



**Figure 3.1:** GUI samples from the WebColor [3] training set

## 3.2 Filling the Gaps in GUI Datasets with FLUID

Despite the efforts made toward the creation of such large and refined datasets like Klarna [2] and WebColor [3], they could be further enhanced to better accommodate the needs of both designers and AI researchers. Specifically:

- If a dataset were to be compatible with Figma [1], meaning that the dataset's GUIs are in a format that can be opened and edited within the Figma application, an entire world of new possibilities unfolds. This is because the internal GUI representation used by Figma would then be available for the development of new AI models that could be directly deployed, via plugin, within the Figma environment. These AI models, trained on Figma-compatible GUIs, could seamlessly integrate into the designer's workflow by suggesting new design elements, highlighting potential design pitfalls, and offering optimizations, thereby enhancing productivity and creativity in the design process.
- HTML web pages can sometimes be verbose and poorly organized, making them less suitable for training AI models that want to address issues strictly

related to GUI design. A simplified hierarchical data representation that focuses on the most relevant aspects for GUI design, instead, would make the dataset AI-ready, also removing unnecessary data that could be perceived as "noise" by AI models.

To target these needs, a new dataset derived from WebColor is proposed, called **FLUID**, Figma Layout User Interface Dataset. WebColor is selected as the source dataset because it retains the advantages of Klarna by providing both MHTML files and screenshots, allowing for flexibility in data management, while also offering a more refined structure where unnecessary elements are removed.

Moreover, since the final objective of this work is to propose an AI model for multimodal GUI representation learning to support the development of new AI models for downstream tasks, the Rico [4] dataset introduced in Section 2.2.3 is not considered. The dataset lacks individual images from each GUI and does not provide sufficiently precise bounding boxes for cropping and storing images in separate files. As the availability of images is essential in a multimodal context, Rico is discarded.

## 3.3 Data Collection

To derive FLUID from WebColor, the MHTML format was selected because it is the most comprehensive one, including HTML, CSS files and additional content (e.g., images).

From visual inspection of the data, it was observed that cookie notifications significantly impact the visibility of product information on web pages. Figure 3.2a shows a typical example of a web page where the product details are partially obscured by cookie notifications. To address this issue, **the `<div>` tag associated with cookie notifications is removed from the HTML files, resulting in a clearer display of the page content**, as it can be seen in Figure 3.2b.

Next, to convert web pages to a format that is compatible with Figma [1], the Figma plugin **Builder.io-Figma to HTML, React and more** [59] can be used. It allows to:

- Generate a Figma-compatible JSON file for a GUI starting from the web page files (i.e., HTML, CSS, images).
- Import the previously generated JSON file into Figma.

A local web server in the directory containing the plugin module and another in the dataset subdirectory are manually initiated during the setup process. These servers are configured to handle cross-origin requests on distinct ports.

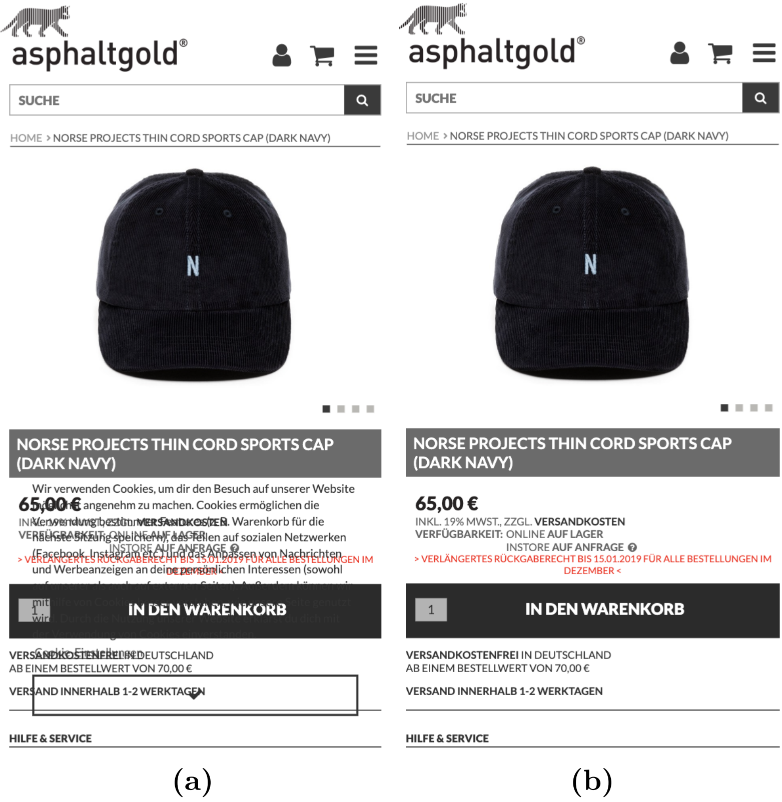



**Figure 3.2:** Visual aspect of a WebColor [3] web page with and without the cookie notification. (a) Web page with the cookie notification partially obscuring product information. (b) Web page after removing the cookie notification, showing clear product information

Then, using a JavaScript (JS) script, the viewport size of the browser page is configured to simulate the screen dimensions of a smartphone, and **the plugin module is dynamically injected in the HTML page and executed**.

**The plugin execution results in a JSON file** that is further manipulated by the JS script.

Specifically, when importing a JSON file in Figma using the original plugin, images are correctly visualized only if their source is a web URL. Since the dataset images are saved locally and the plugin does not support the visualization of images with local URLs in Figma, to ensure full compatibility **images are converted, using the JS script, to Base64 encoded strings**. Consequently, **the plugin is modified to be able to render not only images retrieved from the web, but also images encoded as Base64 strings**. In this way, images can be correctly imported in Figma.

SVGs are modified by the JS script using the Cheerio [60] library to enhance compatibility. **The `<use>` tags have been replaced with their actual referenced elements, directly in the `<svg>` tag**, maintaining any necessary `viewBox` attributes for proper scaling. This process results in a self-contained, updated SVG string that is better suited for environments that may not support external references, like Figma.

Finally, the resulting GUI representation is saved into a new JSON file. The entire procedure is repeated for all the samples in the WebColor dataset.

Table 3.1 provides a description of the main GUI nodes' attributes in the resulting Figma-compatible JSON files, while Table 3.2 reports the description of GUI nodes' categories, which are the values that the `type` attribute, shown in Table 3.1, can assume.

| **Attribute** | **Description** |
| --- | --- |
| `type` | Indicates the type (category) of the Figma node |
| `x` | Defines an x-axis coordinate in the user coordinate system. The x is relative to the parent |
| `y` | Defines a y-axis coordinate in the user coordinate system. The y is relative to the parent |
| `width` | Sets an element's width |
| `height` | Sets an element's height |
| `clipsContent` | Automatically crops any content that goes beyond the bounds of a component. This means that whatever is outside the component's bounds won't be visible in the design |
| `r`, `g`, `b`, `a` | Defines the intensity of the color, with a value between 0 and 255, and opacity |
| `opacity` | Sets the opacity of an element |
| `url` | Contains the path of the image |
| `scaleMode` | How the image is positioned and scaled within the layer |
| `intArr` | Contains the Base64 image representation. This is important since the Figma plugin cannot fetch images from local resources |
| `name` | The name of the component. This is useful to specify a data-layer attribute to make things more debuggable when inspecting the sublayers of a widget |
| `characters` | Textual content |

Continued on next page

**Table 3.1 – continued from previous page**

| Name | Description |
|---|---|
| `blendMode` | Determines how the color of this shadow blends with the colors underneath it |
| `textCase` | Overrides the case of the raw characters in the text node |
| `fontSize` | The size of the font |
| `fontFamily` | The family of the font |
| `textAlignHorizontal` | The horizontal alignment of the text with respect to the Text node |
| `unit` | Indicates the unit of measure of `fontSize` |
| `strokeWeight` | Sets the width of the stroke used for points, lines, and the outlines of shapes |
| `radius` | The blur radius of the shadow |
| `svg` | SVG elements |
| `position` | Set based on the element's computed `position` property, specifically if it is either absolute or fixed, indicating that the element is positioned out of the normal document flow and placed at a specific location |
| `heightType`<br>`widthType` | Determined by whether the element's height or width are fixed or should shrink to fit content. The types are also influenced by the element's `display` property (inline elements are set to "shrink") and parent alignment properties (e.g., textAlign, justifyContent) |
| `topLeftRadius`<br>`topRightRadius`<br>`bottomRightRadius`<br>`bottomLeftRadius` | Used to make round corners |
| `horizontal` | Defines how a layer behaves when resizing the frame along the x axis |
| `vertical` | Defines how a layer behaves when resizing the frame along the y axis |

**Table 3.1:** Description of GUI elements' attributes in FLUID JSON files

| **Category** | **Description** |
|---|---|
| `FRAME` | Used to create structured containers for UI elements |
| `IMAGE` | Allows to insert images within designs |
| `RECTANGLE` | Used to draw rectangular shapes |
| `DROP_SHADOW` | Adds a shadow effect to objects |
| `TEXT` | Enables adding and styling text |
| `SOLID` | Used for creating solid color fills |
| `SVG` | Incorporates SVG graphics into designs |

**Table 3.2:** Description of GUI nodes' categories in FLUID JSON files

The Code Block 3.1 shows an example of the FLUID hierarchical representation, excerpted from a JSON file.

These JSON files can be directly imported in Figma [1] by using the Figma Builder.io-Figma to HTML, React and more [59] plugin's interface. Figure 3.3 shows the plugin interface. The red rectangle highlights the position of a clickable link to import the previously generated JSON files.

Figure 3.4b shows the accurate reconstruction when importing a FLUID GUI in Figma, mirroring the corresponding WebColor web page visible in Figure 3.4a. It's important to note, however, that while this instance was successful, variations in GUI complexity might affect reconstruction fidelity since the process of transforming HTML to a Figma-compatible format is a best-effort process. An example of a sub-optimal reconstruction is shown in Figure 3.4d, for the WebColor web page in Figure 3.4c. The overall structure of the GUI is preserved, but there are some missing elements (e.g., missing cart) and slight misalignments in visual appearance (e.g., the "-20%" discount text).

```
{
  "layers": [
    {
      "type": "FRAME",
      "width": 375,
      "height": 829,
      "x": 0,
      "y": 0,
      "children": [
        {
          "type": "FRAME",
          "clipsContent": false,
          "x": 0,
          "y": 0,
          "width": 375,
          "height": 44,
          "backgrounds": [],
          "children": [
            {
              "type": "RECTANGLE",
              "x": 0,
              "y": 0,
              "width": 375,
              "height": 44,
              "fills": [
                {
                  "type": "SOLID",
                  "color": {
                    "r": 0.9921568627450981,
                    "g": 0.8509803921568627,
                    "b": 0.3607843137254902
                  },
                  "opacity": 1
                }
              ],
              "topLeftRadius": 0,
              "topRightRadius": 0,
              "bottomRightRadius": 0,
              "bottomLeftRadius": 0,
              "constraints": {
                "horizontal": "SCALE",
                "vertical": "MIN"
              }
            }
          ]
        }
      ]
    }
  ]
}
```

**Code Block 3.1:** Example of the FLUID hierarchical representation, excerpted from a JSON file

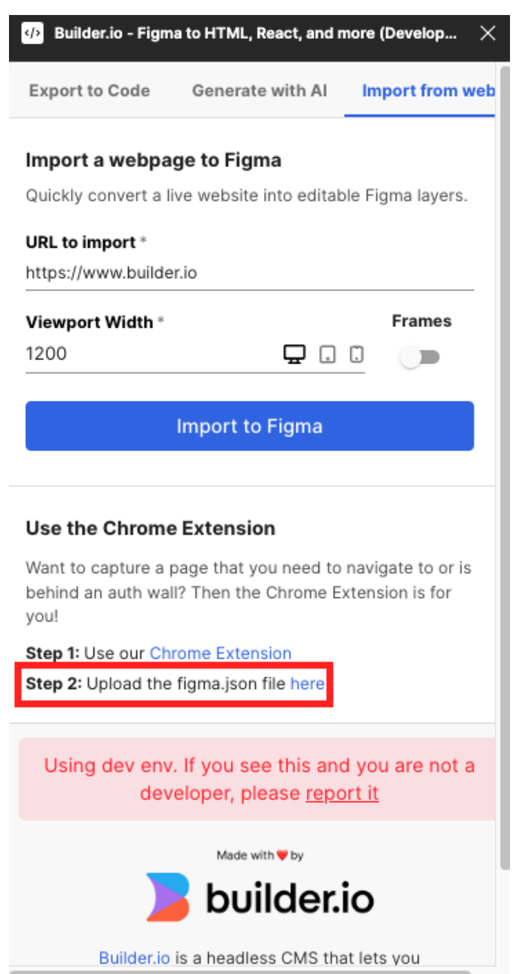



**Figure 3.3:** Figma plugin Builder.io-Figma to HTML, React and more [59] interface. The red rectangle highlights the position of a clickable link to import a JSON file

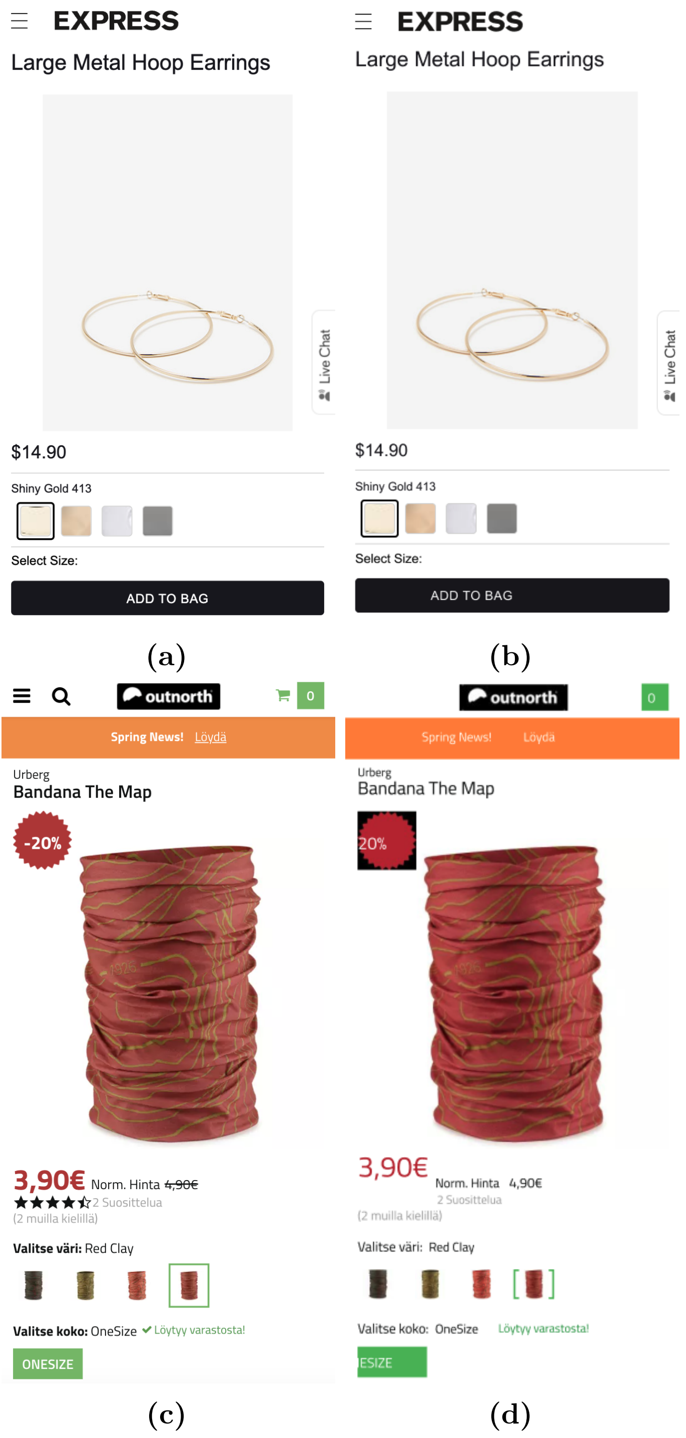



**Figure 3.4:** Visual comparison between two WebColor web pages and their corresponding FLUID GUIs imported in Figma. (a), (c) WebColor web pages. (b), (d) FLUID GUIs imported in Figma

## 3.4 Data Analysis and Cleaning

The JSON files compatible with Figma are now analyzed to further refine the FLUID dataset.

Figure 3.5 illustrates the distribution of the total number of GUI elements per file, which resembles a normal distribution with a peak around 40, suggesting that most files contain approximately 40 elements.

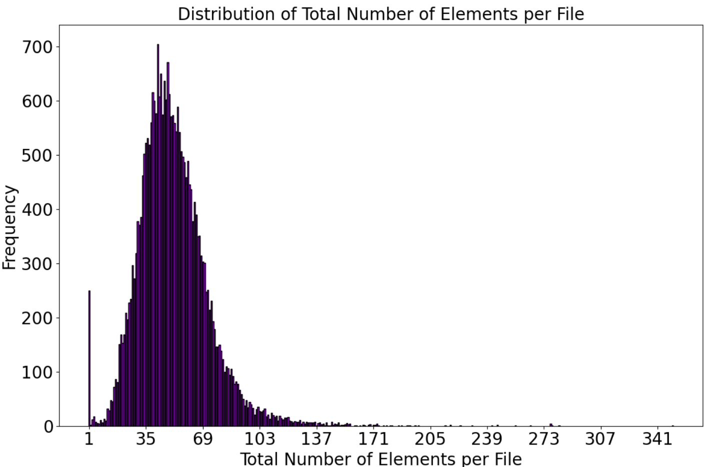


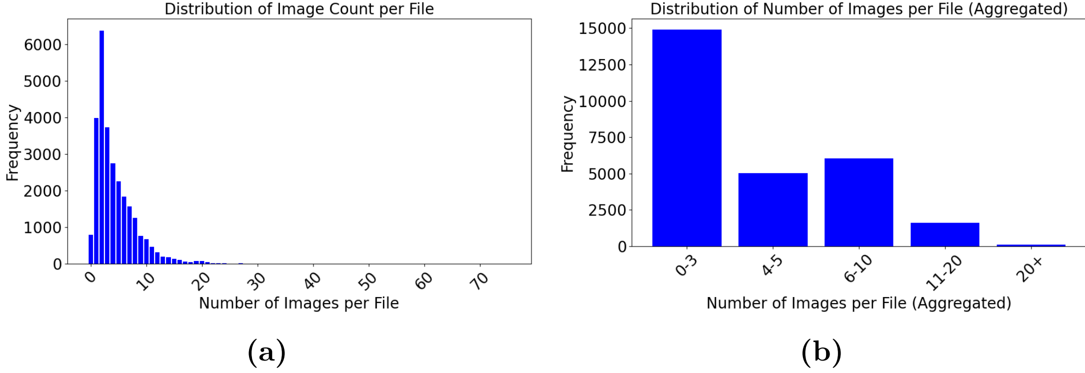

**Figure 3.5:** Distribution of the total number of elements per file in FLUID

There are also some outliers, with files containing a high number of elements and others containing as few as one element. These extremes may result from earlier data cleaning steps, such as the removal of cookies elements.

Figure 3.6 shows the distribution of the number of images per file; in Figure 3.6b the number of images are grouped into aggregated intervals.

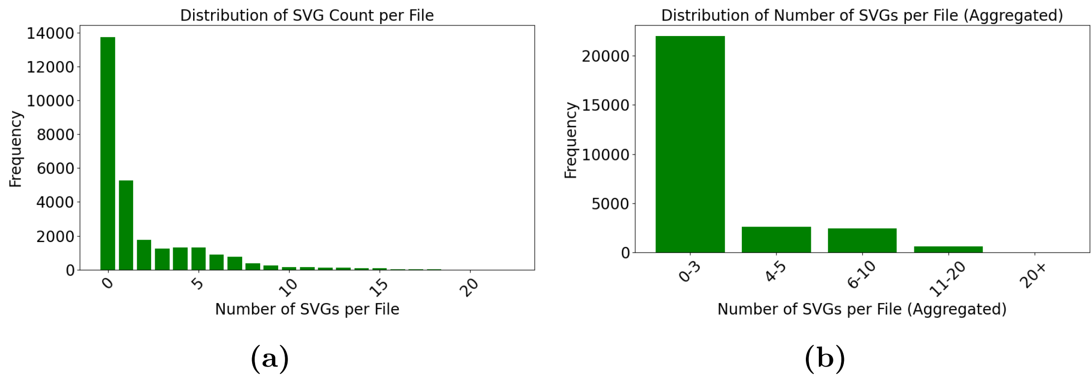
The distribution is concentrated on the left, meaning that most files contain a small number of images, with the frequency decreasing as the number of images per file increases.

Figure 3.7 shows the distribution of the number of SVGs per file.




(a) (b)

**Figure 3.6:** Distribution of the number of images per file in FLUID. (a): distribution of the number of images per file. (b): distribution of the number of images per file as aggregated intervals



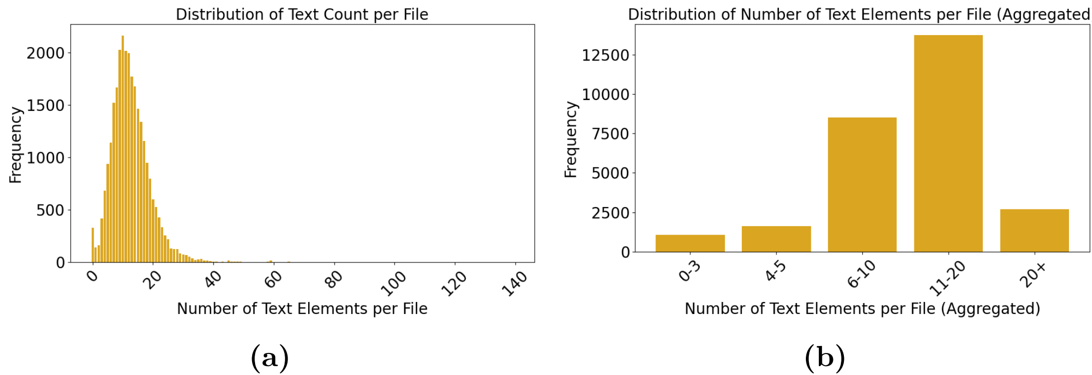

(a) (b)

**Figure 3.7:** Distribution of the number of SVGs per file in FLUID. (a): distribution of the number of SVGs per file. (b): distribution of the number of SVGs per file as aggregated intervals

A significant number of files contains 0 SVGs and very few files contain more than 7 SVGs, pointing out that files that contain a lot of SVGs are rare.

Figure 3.8 and Figure 3.9 offer a view of the distribution of text elements and their length within the dataset.

The distribution of the number of text elements (see Figure 3.8) inside each file is highly concentrated on the left with a peak around 10. There is a rapid decline in frequency as the number of text elements per file increases, indicating that most files contain relatively few text elements. Very few files contain more than 40 text elements, highlighting that text-heavy files are very rare. The distribution of text lengths (Figure 3.9) displays a right-skewed distribution, with most texts containing between 0 to 300 words, indicating that shorter texts are predominant




**(a)** **(b)**

**Figure 3.8:** Distribution of the number of textual elements per file in FLUID. (a): distribution of the number of textual elements per file. (b): distribution of the number of textual elements per file as aggregated intervals

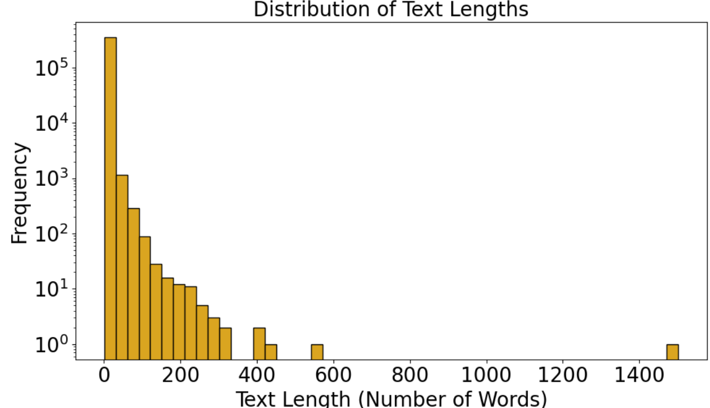



**Figure 3.9:** Distribution of the length of textual elements, in number of words, in FLUID

in the dataset. Few texts extend up to 1,400 words, highlighting outliers and suggesting a potential need for special handling of such values.

Further analysis has been conducted on the values of the attributes `x`, `y`, `width`, `height`, whose results are shown in Figure 3.10. It's important to note that the `x` and `y` attributes in FLUID actually specify the position relative to the parent element, while the distributions presented in Figures 3.10a and 3.10b are produced by converting relative positions to absolute positions with respect to the overall GUI layout.

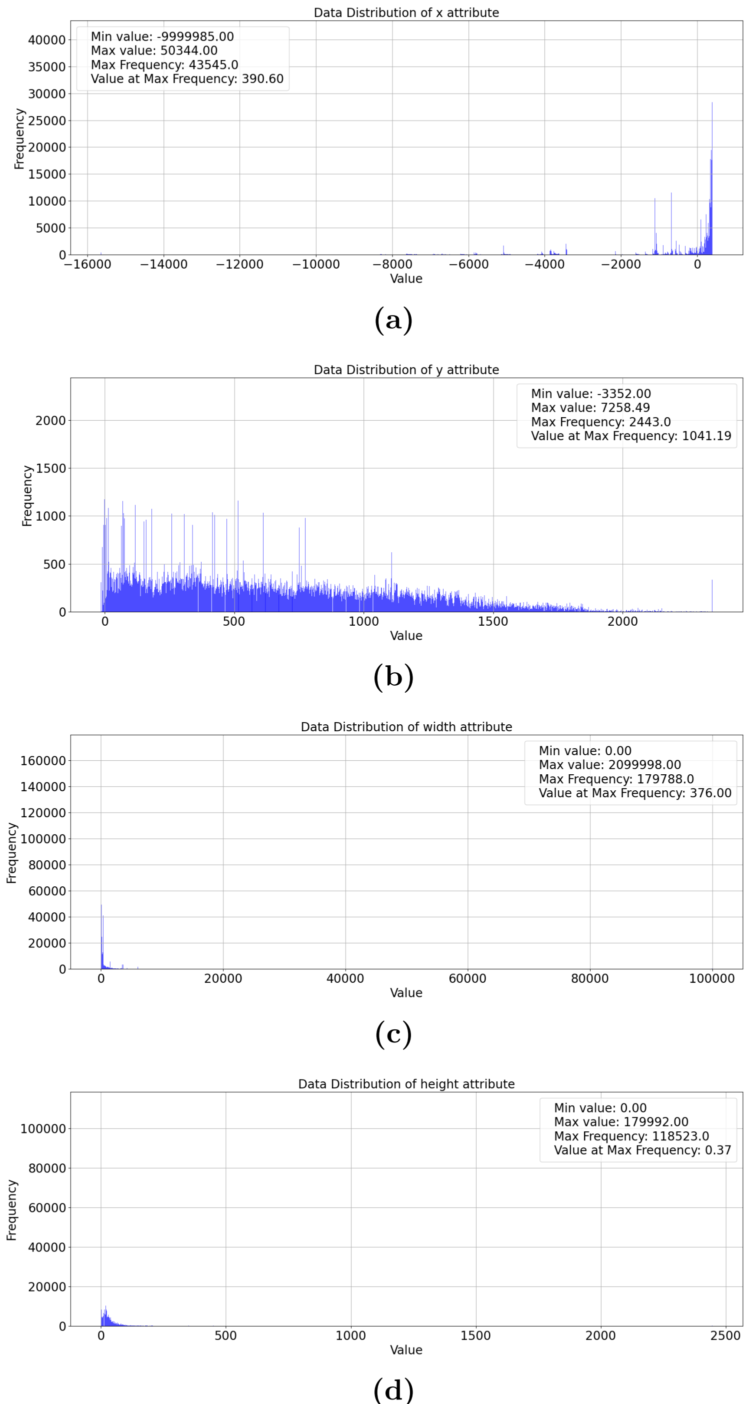



**Figure 3.10:** Distribution of the attributes `x`, `y`, `width` and `height` in FLUID. (a) Distribution of the attribute `x` (absolute). (b) Distribution of the attribute `y` (absolute). (c) Distribution of the attribute `width`. (d) Distribution of the attribute `height`

The FLUID interfaces have a size of 375 $\times$ 1,400 pixels. From Figure 3.10, it is evident that `x` is often less than 0, while `y` is often greater than 1,400, suggesting there are many UI elements that extend beyond the limits imposed by the FLUID GUI size.

In fact, Figure 3.11 reports an example of a `FRAME` element whose `width` is greater than 375. This makes the element overflow the GUI opened in Figma.

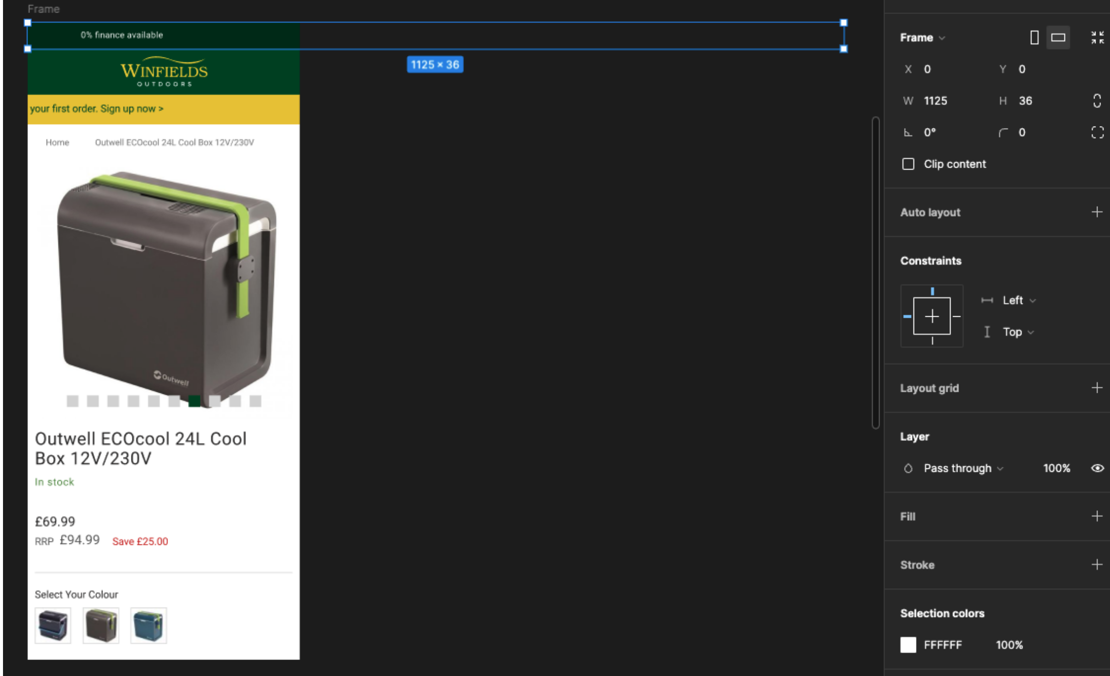



**Figure 3.11:** Example of a FLUID GUI opened in Figma [1], where `width` is greater than 375

Despite extending beyond the GUI limits, these components are kept since it is possible to compute the correct positions for the elements they contain.

Figure 3.12 illustrates the relative frequencies of the main attributes reported in Table 3.1 and some other attributes that were not reported in the table.

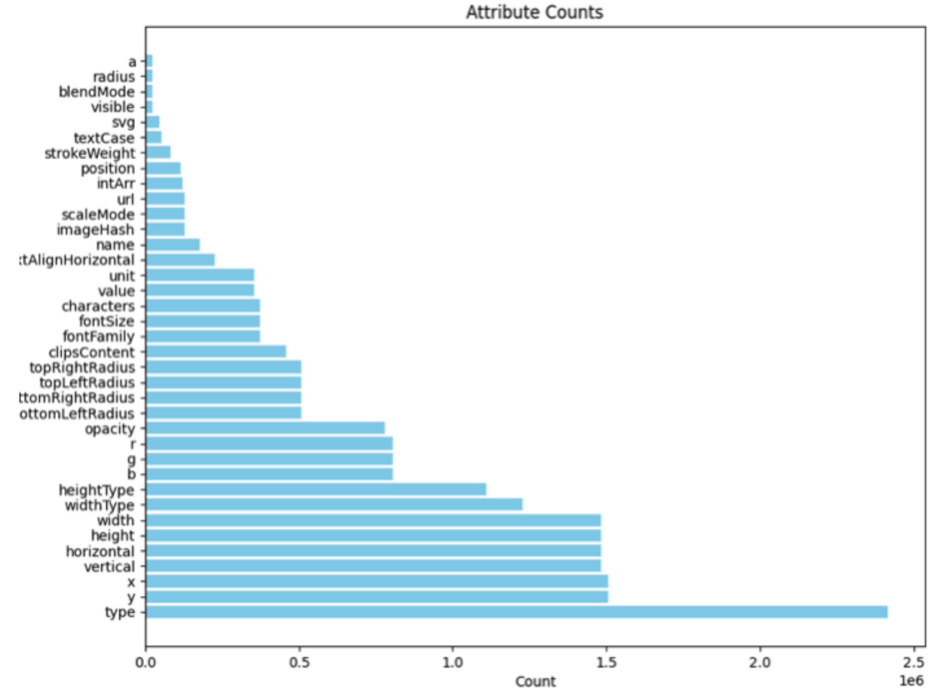



**Figure 3.12:** Distribution of GUI elements' attributes in FLUID dataset

For the scope of this work, attention will be focused solely on static elements; consequently, **properties such as `horizontal`, `vertical`, `heightType`, `widthType`, `textAlignHorizontal` and `clipsContent`, which are related to dynamic behavior, are deleted**.

**Certain attributes, such as `a`, `radius`, `textCase`, and `strokeWeight`, exhibit low usage frequencies and, for this reason, are discarded**.

Additionally, **the attributes `blendMode` and `name` consistently assume the same values over different GUI elements and are therefore eliminated**.

**`imageHash` is always `None` and, for this reason, it is removed**.

**The attributes `topLeftRadius`, `topRightRadius`, `bottomRightRadius`, `bottomLeftRadius` and `opacity` do not significantly contribute to the aspect of the user interface, so they are deleted**. Figure 3.13 demonstrates the effect of removing the `topLeftRadius`, `topRightRadius`, `bottomRightRadius`, and `bottomLeftRadius` attributes. Initially, the carousel indicators are round, as shown in Figure 3.13a, but after the removal of these attributes, they become squared, as illustrated in Figure 3.13b.

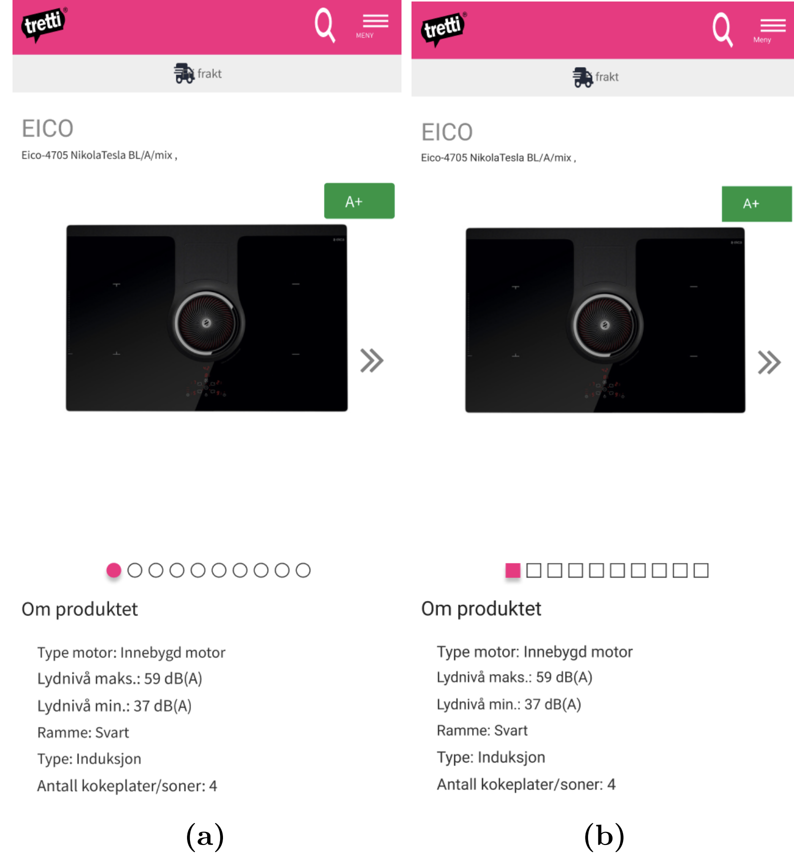



**Figure 3.13:** Effect of removing the attributes `topLeftRadius`, `topRightRadius`, `bottomRightRadius`, and `bottomLeftRadius` from FLUID GUIs. (a) The carousel indicators are round when these attributes are present. (b) After the removal of these attributes, the carousel indicators become squared

In many cases, the original WebColor HTML pages include `<a>` elements with a dummy destination URL. An example is reported in Code Block 3.2. The conversion to JSON transforms these elements to `RECTANGLE` nodes that, when opened in Figma, appear as **grey boxes that negatively impact the visual layout**, as shown in Figure 3.14. Therefore, **such `RECTANGLE` nodes are removed from FLUID JSON files to improve the Figma visualization**.

```
<a href="dummy" id="element-8" klarna-ai-label="Cart">
    <img id="element-9" src="inline-0.svg">
</a>
```

**Code Block 3.2:** Example of an `<a>` HTML tag in WebColor dataset with a dummy destination URL

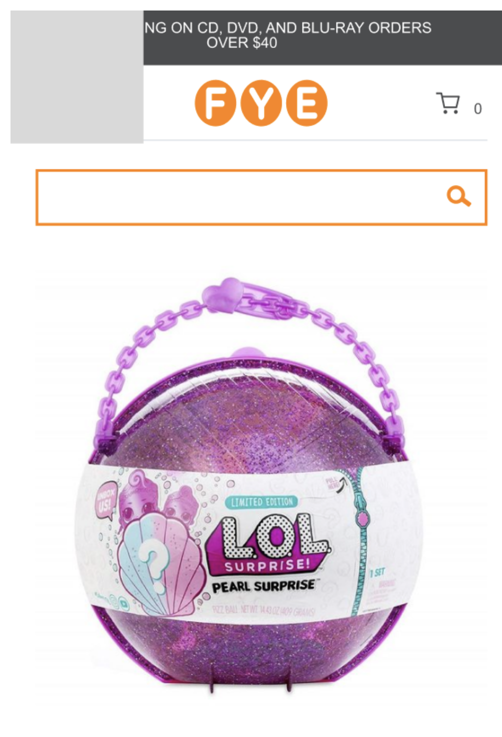



**Figure 3.14:** Example of a grey box negatively impacting the visual layout of a FLUID GUI opened in Figma

All of the categories reported in Table 3.2 are strictly mandatory for Figma to correctly render the GUIs, except for `DROP_SHADOW`, which is optional and only adds style to objects. Therefore, **`DROP_SHADOW` type nodes are deleted because they don't significantly influence the visual appearance**.

`RECTANGLE` always contains a node of type `TEXT` or a node of type `IMAGE`, since it is just a wrapper for these two categories. However, being a category mandatory to render GUIs in Figma, it cannot be removed.

The final distribution of elements per file after these data cleaning steps is shown in Figure 3.15.

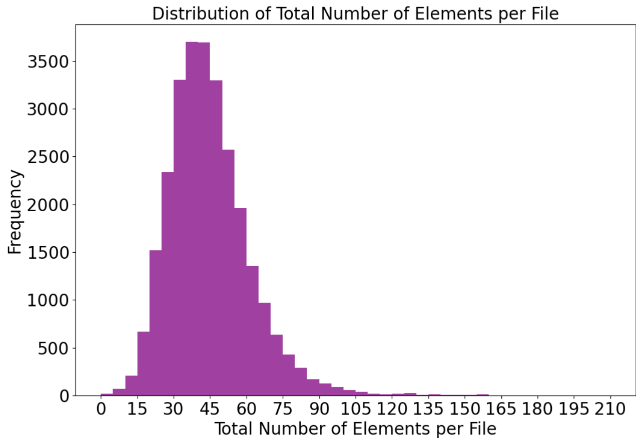



**Figure 3.15:** Distribution of the total number of elements per file in FLUID after data cleaning

Compared to Figure 3.5, the reported data cleaning steps have effectively reduced the presence of outliers. However, to further ensure a balanced dataset and potentially better training dynamics, it has been decided that **the JSON files of GUIs that include less than 5 elements or more than 90 elements will be deleted**.

The final dataset now contains 41,337 samples divided as:

- 25,825 training samples.
- 2,972 validation samples.
- 12,540 test samples.

## 3.5 Conclusion

Finally, Figure 3.16 reports a visual representation for the operations performed on the WebColor dataset to build FLUID. The operations can be summarized as follows:

1. Removal of `<div>` tags associated with cookies notifications.
2. Figma plugin injection in HTML page.
3. Figma plugin execution to extract a JSON file.
4. Conversion of images to Base64 encoded strings and modification of Figma plugin to be able to render these images encoded as Base64 strings.

5. Resolved references of `<use>` tags for SVGs.
6. Removal, after data analysis, of some GUI elements' attributes and categories.
7. Removal of grey `RECTANGLE` nodes caused by a dummy destination URL for WebColor `a` elements.
8. Removed JSON files corresponding to GUIs with less than 5 or more than 90 elements.

Overall, FLUID appears to be both well-structured and with a good degree of compatibility with Figma, laying the foundations for direct integration in Figma of AI models trained on such dataset.

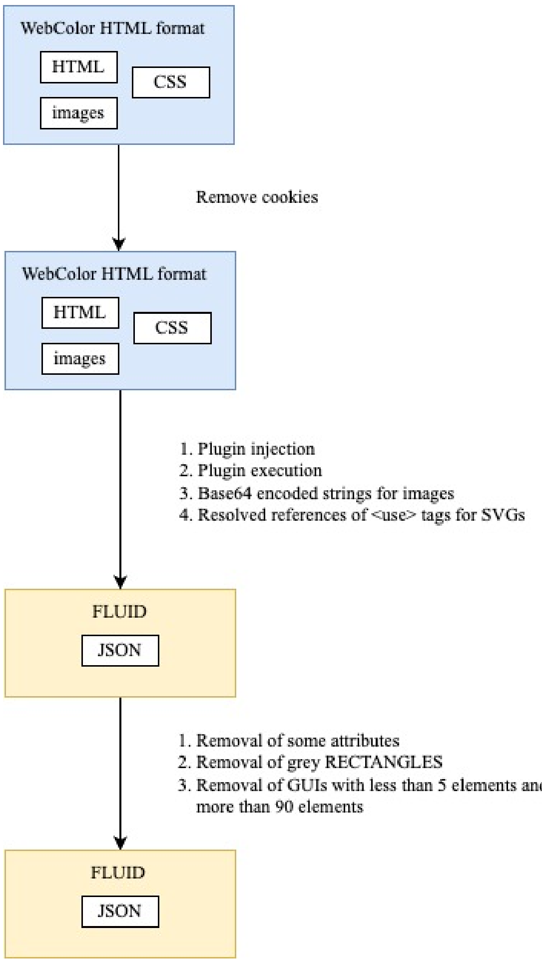



**Figure 3.16:** Operations performed on the WebColor dataset to build FLUID

# Chapter 4

# Methodology

A Graphical User Interface (GUI) contains a series of elements. The goal of this work is to develop an Artificial Intelligence (AI) model to learn an alternative representation for these elements that can be used for efficient training of downstream AI models.

To this end, the chosen architecture is a **Vector Quantized-Variational Autoencoder** (VQ-VAE) [48], a variant of the standard Variational Autoencoder (VAE) [15] designed to learn discrete latent representations.

This chapter first outlines, in Section 4.1, the model architecture, the learning procedure and the implementation details in the simple scenario of elements characterized only by their categories and bounding box coordinates.

Then, Section 4.2 provides the same information but in a complex multimodal framework that can serve a wide variety of downstream real-world applications.

## 4.1 Bounding Box and Category Encoded Representation Learning

A GUI $x \in X$ can be described by the list of the contained elements $x = [g_1, g_2, ..., g_n]$ where $g_i$ is the $i^{th}$ element and $n$ is the total number of elements contained in each GUI. Each element $g_i$ is identified by its **bounding box** $b_i$, with $b_i \in B$, specified by the $x$ and $y$ coordinates of the upper-left and lower-right corners, and its **category** $c_i$ with $c_i \in C$, therefore $g_i = [b_i^{x_0}, b_i^{y_0}, b_i^{x_1}, b_i^{y_1}, c_i]$. The notation can be simplified as $g_i = [b_i, c_i]$.

Given this formulation, **the objective is to develop, train and test an AI model suitable for encoded representation learning in this simple scenario, where GUI elements are characterized only by their categories and bounding box coordinates**.

### 4.1.1 Model Architecture

The **VQ-VAE** [48] is used to learn a discrete latent representation for GUI elements. This learned representation can be particularly useful for downstream tasks where discrete latent variables are beneficial, such as discrimination, clustering of similar patterns into discrete groups and generation of new components. Unlike standard Variational Autoencoders (VAEs) [15], which operate over continuous latent variables, VQ-VAEs map continuous latent vectors to a finite set of discrete vectors, known as *codebook*. This results into latent representations that are more interpretable and can also prevent issues such as posterior collapse, a common issue in traditional VAEs.

The VQ-VAE model architecture includes three key modules:

- **Encoder**: transforms the input into a continuous latent space.
- **Vector Quantizer**: maps the continuous latent representation into a discrete representation.
- **Decoder**: reconstructs the input from the discrete latent space.

The proposed VQ-VAE architecture includes transformer-based self-attention mechanisms, which are particularly effective for capturing long-range dependencies in sequential data, such as GUI components. This makes it well-suited for GUI layouts, which exhibit spatial and structural relationships that need to be modeled effectively. Figure 4.1 shows the proposed VQ-VAE model architecture.

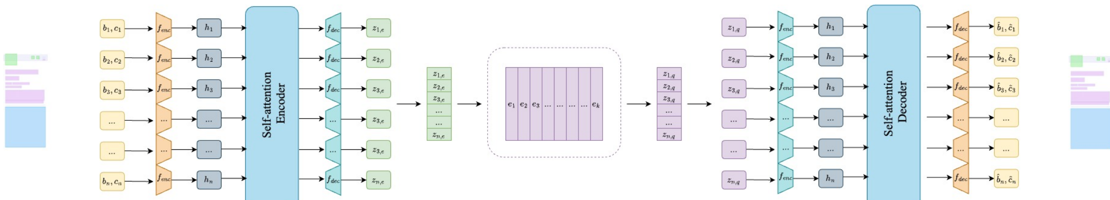



**Figure 4.1:** Proposed VQ-VAE model architecture

In the following, the three modules (i.e., encoder, vector quantizer, decoder) will be presented in detail.

#### Encoder

The architecture of the encoder module is represented in Figure 4.2.

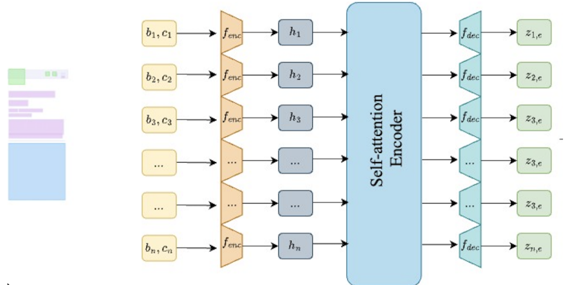



**Figure 4.2:** Proposed VQ-VAE encoder module architecture

The encoder $q_\phi(Z_e|B, C)$ takes as input the sequence of elements $x = [g_1, g_2, ..., g_n]$ of a GUI, where $g_i = [b_i, c_i]$, and uses a **Multi-Layer Perceptron** (MLP), $f_{enc}(b_i; c_i; \phi)$, to project the input into a $d$-dimensional space.

Given that the input contains a series of GUI components, it is essential to include information regarding the position of each element within the sequence. The MLP is followed by a **Transformer** [13]. However, transformers do not inherently encode the order of elements; therefore, **learnable positional embeddings**, $p_i$, are added to each element's hidden representation [61], as described in Equation 4.1.

$$h_i = f_{enc}(g_i; \phi) + p_i \tag{4.1}$$

where $g_i = [b_i^{x_0}, b_i^{y_0}, b_i^{x_1}, b_i^{y_1}, c_i]$.

The modified hidden representation $h_i$ is then passed to a self-attention based mechanism for further refinement, as shown in Equation 4.2.

$$l_i = Transformer(h_i; \phi) \tag{4.2}$$

The choice of using a transformer is motivated by its ability to capture long-range dependencies between elements in the sequence. In the context of GUI layouts, this means understanding relationships such as the spatial organization of components, as well as how elements might visually interact.

After processing through the transformer layers, the sequence is projected into the latent space suitable for quantization using another MLP.

$$z_{i,e} = f_{dec}(l_i; \phi) \tag{4.3}$$

This projection ensures that the continuous latent representation is of the correct dimensionality to be mapped into a discrete latent representation.

### Vector Quantizer

The conversion from a continuous latent space to a discrete one is performed using a Vector Quantizer (VQ), illustrated in Figure 4.3.

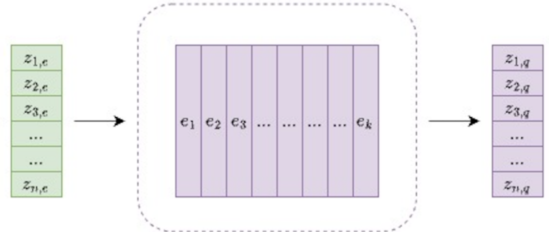



**Figure 4.3:** Proposed VQ-VAE Vector Quantizer module

To discretize the output of the encoder, a codebook consisting of $K$ learned vectors $e_k \in \mathbb{R}^D$ for $k = 1, ..., K$ is defined. For every latent vector $z_{i,e}$ of the encoder output, the Euclidean distances $d_{i,k}$ from every codebook vector $e_k$ are computed following the Equation 4.4.

$$d_{i,k} = ||z_{i,e} - e_k||_2^2 \tag{4.4}$$

For every latent vector $z_{i,e}$, the index $k_i$ of the closest discrete vector $e_{k_i}$ of the codebook is identified based on the Euclidean distances determined in the previous step, following the procedure in Equation 4.5.

$$k_i = arg \min_k d_{i,k} \tag{4.5}$$

The latent vector $z_{i,e}$ is finally discretized by replacing it with the nearest discrete vector from the codebook, $e_{k_i}$, as in Equation 4.6.

$$z_{i,q} = e_{k_i} \tag{4.6}$$

To maintain stability during training, the codebook vectors are updated using **Exponential Moving Average** (EMA) [62]. The update via EMA allows the codebok to evolve gradually over time, avoiding abrupt changes that could destabilize the learning process.

Two information are stored for the EMA procedure:

1. **Embeddings** $e_k$ for each $k \in K$.
2. **Counts** $c_k$ for each $k \in K$, measuring the number of continuous latent representations that have $e_k$ as its nearest neighbor.

The counts $c_k$ are updated over a mini-batch of size $m$, **at training time only**, as described in Equation 4.7.

$$c_k \leftarrow \lambda c_k + (1 - \lambda) \sum_m 1[z_{i,q} = e_k] \tag{4.7}$$

Then, the embeddings $e_k$ are consequently updated as in Equation 4.8.

$$e_k \leftarrow \lambda e_k + (1 - \lambda) \sum_m \frac{1[z_{i,q} = e_k]\, z_{i,e}}{c_k} \tag{4.8}$$

In Equations 4.7 and 4.8, $1[.]$ is the indicator function and $\lambda$ is a decay parameter. Small values of $\lambda$ allow faster updates for the codebook vectors, to better adapt to recent data, while large values make the updates more conservative, which can lead to smoother and more stable updates but might make the model slower to adapt to new patterns in the data.

### Decoder

The decoder $p_\theta(\hat{B}, \hat{C}|Z_q)$, whose architecture is shown in Figure 4.4, is responsible for reconstructing the original input from the quantized latent representations. **The architecture of the decoder mirrors the encoder one**.

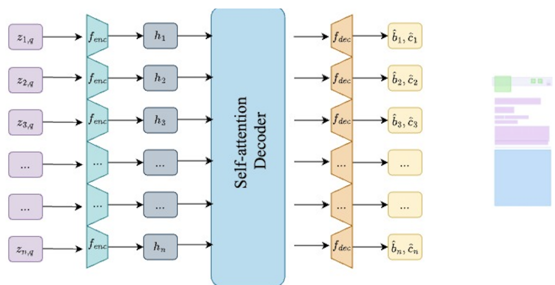



**Figure 4.4:** Proposed VQ-VAE decoder module architecture

The quantized latent vectors $z_{i,q} \in \mathbb{R}^D$ first pass trough an MLP, $f_{enc}(z_{i,q}; \theta)$, to project them into a $d$-dimensional space, ensuring the correct dimensionality for the subsequent processing step. As done for the encoder, the hidden input is summed up with learnable positional embeddings, $p_i$, allowing to retain information about the original order in the sequence. These two operations are described in Equation 4.9.

$$h_i = f_{enc}(z_{i,q}; \theta) + p_i \tag{4.9}$$

The sequence $h_i$ is then passed to a *non-autoregressive* transformer-based decoder, as shown in Equation 4.10.

$$l_i = Transformer(h_i; \theta) \tag{4.10}$$

The transformer helps the VQ-VAE decoder to consider the relationships between different parts of the latent sequence during reconstruction. Additionally, **the decoder is made non-autoregressive to generate all outputs in parallel, effectively improving the efficiency of the decoding process**.

After the transformer layers, the output $l_i$ passes through a final MLP to project it back into the original input space, ensuring that the output has the same dimensionality and structure as the original input. The result of this operation, reported in Equation 4.11, is the reconstructed sequence of elements $\hat{x} = [\hat{g}_1, \hat{g}_2, ..., \hat{g}_n] \in \hat{X}$, expressed in terms of its elements $\hat{g}_i = [\hat{b}_i^{x_0}, \hat{b}_i^{y_0}, \hat{b}_i^{x_1}, \hat{b}_i^{y_1}, \hat{c}_i]$, also written as $\hat{g}_i = [\hat{b}_i, \hat{c}_i]$.

$$\hat{g}_i = f_{dec}(l_i; \theta) \tag{4.11}$$

### 4.1.2 Learning Procedure

The total loss for training the proposed VQ-VAE model consists of two terms: the **reconstruction loss** and the **commitment loss**.

In this work, the reconstruction loss $\mathcal{L}_{reconstruction}$ has two parts: a **Mean Squared Error** (MSE) **loss** for reconstructing the bounding boxes and a **categorical Cross-Entropy loss** for predicting the element categories. The reconstruction loss is formulated as in Equation 4.12.

$$\mathcal{L}_{reconstruction} = ||b_i - \hat{b}_i||_2^2 - \alpha \sum_{j=0}^{nc-1} c_{i,j} \log \hat{c}_{i,j} \tag{4.12}$$

where $||b_i - \hat{b}_i||_2^2$ represents the MSE loss computed on the bounding box coordinates, and the second term $-\alpha \sum_{j=0}^{nc-1} c_{i,j} \log \hat{c}_{i,j}$ represents the categorical cross-entropy loss computed using the predicted categories, where $nc$ is the total number of categories for GUI elements and $\alpha$ is a scaling factor that balances the two loss components.

Then, the commitment loss $\mathcal{L}_{commitment}$, formally defined in Equation 4.13, is used to further encourage the encoder module to produce latent vectors that are close to the codebook vectors.

$$\mathcal{L}_{commitment} = ||z_{i,e} - sg(z_{i,q})||_2^2 \tag{4.13}$$

In Equation 4.13, $sg(z_{i,q})$ stands for the *stop-gradient* operation, defined as in Equation 4.14.

$$sg(z_{i,q}) = \begin{cases} z_{i,q} & \text{forward pass} \\ 0 & \text{backward pass} \end{cases} \tag{4.14}$$

This operator prevents gradients from flowing through $z_{i,q}$ during backpropagation, ensuring that **only the encoder is updated and not the codebook vectors**.

Unfortunately, the operation described in Equation 4.5 is not differentiable and does not allow gradients to flow through the quantization process during backpropagation. Therefore, to allow the encoder to receive gradients from both the reconstruction and commitment losses, the **straight-through estimator** [63] technique is applied. This approach treats the quantization step as an identity function during the backward pass, allowing gradients to flow directly through the non-differentiable quantization process. Specifically, while the encoder's output is quantized by assigning it to the nearest codebook vector during the forward pass, the gradients are directly passed from the quantized representation $z_{i,q}$ to the continuous encoder output $z_{i,e}$ during the backward pass. This trick allows the encoder to be optimized despite the non-differentiability of the quantization step.

So, the total loss is a weighted sum of the reconstruction and commitment losses, as illustrated in Equation 4.15.

$$\mathcal{L} = \mathcal{L}_{reconstruction} + \beta \mathcal{L}_{commitment} \tag{4.15}$$

where $\beta$ is a hyperparameter that controls the relative importance of the commitment loss.

Finally, the optimization strategy can be summarized as follows:

- The **decoder** is optimized by minimizing the reconstruction loss, focusing on reconstruction of bounding boxes and prediction of GUI elements categories.
- The **encoder** is optimized by minimizing both the reconstruction loss and the commitment loss, ensuring that the vectors produced by the encoder remain close to the codebook vectors.
- The **embedding space** (i.e., the codebook) is updated using Exponential Moving Average [62], ensuring smooth updates of the discrete vectors over time.

### 4.1.3 Implementation Details

The project is developed using Python 3.9 [64], with PyTorch 1.12.1 [65] and CUDA 11.6 [66].

Both model training and evaluation are executed on remote machines of the Paperspace platform [67]. The specific CPU and GPU configurations are not reported, as they vary based on machine availability at any given time.

The proposed VQ-VAE model is trained and evaluated on both FLUID and the Rico [4] dataset. The discussion in Section 3.2 highlighted the unsuitability of the Rico dataset for multimodal GUI representation learning. Regardless, for the simple scenario of bounding box reconstruction and category prediction it is adequate and can actually serve as a reference for comparison with other works.

The following two sections report the specific implementation details for the two datasets.

### FLUID

When loading data from FLUID, the following preprocessing operations are performed:

- GUI elements from the `FRAME` category are essentially wrappers for other GUI components. Being mandatory to render GUIs in Figma [1], it is not possible to permanently remove them from JSON files. However, for the purpose of this work, GUI elements from this class are filtered out on-the-fly to simplify the model training, while still retaining their subnodes in the GUI elements tree. Ideally, the positions of these `FRAME` elements within the GUI elements tree should be stored to reinsert them at a later stage, to ensure compatibility with Figma. Alternatively, future work could preserve these elements and evaluate the model in a scenario with these additional `FRAME` elements.

- The `RECTANGLE` category is just a wrapper for the `TEXT` and `IMAGE` categories, but it is mandatory to render GUIs in Figma, so it cannot be permanently removed from JSON files. However, for the purpose of this work, GUI elements from the `RECTANGLE` class are filtered out on-the-fly to simplify the model training, while still retaining their subnodes in the GUI elements tree. This does not impact the compatibility with Figma, considering it is possible to add these wrappers in the context of a post-processing operation.

- The number of GUI element categories is 4. The categories that are considered are `SVG`, `TEXT`, `SOLID`, `IMAGE`, and they are one-hot encoded.

- As reported in Table 3.1, the `x` and `y` positions are relative to the parent element. These values are converted to absolute positions and are min-max normalized to the [0, 1] range according to the FLUID max GUI size (i.e., 375 $\times$ 1,400 pixels).

- Following the discussion below Figure 3.11, there are some GUI elements whose bounding boxes extend beyond the GUI limits. However, for some subnodes of these GUI elements the absolute position meets the constraints imposed by the GUI size. Therefore, GUI elements for which at least a coordinate of

the bounding box falls outside the GUI layout have been filtered out. This operation is restricted to these nodes only, and not to their subnodes, which are evaluated independently.

For the VQ-VAE model, the following configuration is used:

- The model is developed to work on sequences of a fixed size, set to 90 according to the analysis below the Figure 3.15. For GUIs that have a lower number of elements than 90, proper padding is added to reach this length and a masking strategy is used to correctly compute the losses and the metrics.

- The encoder and decoder transformers share the same parameters. Specifically, $d$, the number of expected features in the encoder/decoder inputs, is set to 512, the number of multi attention heads is set to 8, the dimension of the feedforward model is set to 2,048, the number of layers is set to 12.

- For the codebook, $K$ is set to 256, $D$ is set to 64. The codebook vectors are updated via EMA as described in Equations 4.7 and 4.8, with the $\lambda$ parameter set to 0.99 for smooth updates.

The training is conducted using the following parameters:

- The batch size for the training dataloader is 128.

- AdamW [68] is used as the optimizer, with the learning rate fixed to $10^{-5}$.

- The scaling factor $\alpha$ (see Equation 4.12) for the categorical Cross-Entropy loss is set to 0.01.

- The scaling factor $\beta$ (see Equation 4.15) for the commitment loss is set to 0.25.

- The validation step is set to 2. The batch size for the validation dataloader is 1.

- The validation metrics are monitored and the *Early Stopping* algorithm is used to stop the training when no improvements are recorded for a given number of validation steps, that is the patience, set to 30.

- The training is performed indefinitely, until it is stopped by the *Early Stopping* algorithm.

#### Rico Dataset

When loading data from Rico, the following preprocessing operations are performed:

- The $x_0$, $y_0$, $x_1$ and $y_1$ values for positions are already in absolute format, differently from FLUID. These values are min-max normalized to the [0, 1] range according to the Rico max GUI size (i.e., 1,440 × 2,560 pixels).
- The number of GUI element categories is 25, as reported in Section 2.2.3, and they are one-hot encoded.

The VQ-VAE model configuration is unchanged from the one used for FLUID.
The only difference in training with respect to FLUID is the *Early Stopping* patience, set to 25.

## 4.2 Multimodal Encoded Representation Learning

In addition to the bounding box coordinates and element categories used to train the VQ-VAE [48] in the simple scenario described in Section 4.1, this work also explores the inclusion of **text**, **images** and **colors** to enrich the representation of GUI components. **The objective is to go one step further and develop, train and test an AI model for multimodal encoded representation learning of GUIs, capturing a broader range of features that could be exploited by downstream real-world applications**. To address this task, the chosen architecture for the AI model is a VQ-VAE, the same one described in Section 4.1.1 and with the minimal set of changes required to accommodate the needs of the new, extended GUI representations.

### 4.2.1 Multimodal Input Representation

For the textual elements of GUIs, **text embeddings are generated using BERT** (Bidirectional Encoder Representations from Transformers) [61], a pre-trained language model that captures deep contextual information. BERT is well-suited for generating embeddings that reflect the meaning and context of text, which is crucial for understanding the role of textual elements like button labels, headings, or menu items within a GUI.

For each GUI element that contains text, BERT generates a fixed-length embedding $t_i \in \mathbb{R}$ for the text, that is subsequently concatenated with the bounding box and category information of the element. For elements without text, a zero-vector is used as a placeholder. The extended representation for a GUI element can now be formalized as in Equation 4.16.

$$g_i = [b_i^{x_0}, b_i^{y_0}, b_i^{x_1}, b_i^{y_1}, c_i, t_i] \tag{4.16}$$

By leveraging BERT's ability to generate semantically rich text embeddings, the VQ-VAE model is better equipped to capture the role of text in GUI layouts, enabling it to learn more effective representations for components where text plays a significant role.

To represent images in GUIs, **image embeddings are extracted using CLIP** (Contrastive Language–Image Pretraining) [69]. CLIP is a powerful model that generates semantically meaningful embeddings for images based on their visual content. This allows the VQ-VAE model to better capture the relationships between visual elements within the GUI, potentially enhancing its ability to learn representations that reflect the structure and content of graphical layouts.

For each image element in the dataset, CLIP is used to generate a fixed-length embedding $v_i \in \mathbb{R}$. The image embedding is appended to the element representation alongside the bounding box, category and text information. If a GUI element does not contain an image, a zero-vector is used as a placeholder. The updated GUI element representation is reported in Equation 4.17.

$$g_i = [b_i^{x_0}, b_i^{y_0}, b_i^{x_1}, b_i^{y_1}, c_i, t_i, v_i] \tag{4.17}$$

Using CLIP, the VQ-VAE model can leverage the rich relationships between images, improving its capacity to represent GUI components in a way that reflects both their visual appearance and their functional context within the layout.

Color also plays an important role in the visual appearance of GUI components. In this work, **color information is represented by predefined color categories**, such as "red", "blue", or "green". Each GUI element $g_i$ is assigned a category $col_i \in C_{col}$, representing the color of the element. If a GUI element does not include color information, a zero-vector is used as a placeholder. The final extended GUI element representation is shown in Equation 4.18.

$$g_i = [b_i^{x_0}, b_i^{y_0}, b_i^{x_1}, b_i^{y_1}, c_i, t_i, v_i, col_i] \tag{4.18}$$

Following the introduction of these additional features, the final reconstruction output $\hat{g}_i$ of the VQ-VAE model will include bounding box coordinates, element categories, text embeddings, image embeddings, and color categories. This ensures that the model can accurately predict not only the bounding box coordinates and element categories, but also the text embeddings, image embeddings, and color categories, yielding a full reconstruction $\hat{g}_i$, defined in Equation 4.19, for the GUI element $g_i$.

$$\hat{g}_i = [\hat{b}_i^{x_0}, \hat{b}_i^{y_0}, \hat{b}_i^{x_1}, \hat{b}_i^{y_1}, \hat{c}_i, \hat{t}_i, \hat{v}_i, \hat{col}_i] \tag{4.19}$$

This procedure allows for the reconstruction of both visual and semantic aspects of each element.

### 4.2.2 Multimodal Learning Procedure

With the inclusion of the additional features reported in Section 4.2.1 (i.e., text embeddings, image embeddings, and color categories), the learning procedure requires some adjustments to account for these features. In the VQ-VAE model for the simple scenario with only the bounding box coordinates and the GUI element categories (see Section 4.1), the reconstruction loss is computed on this limited set of features, as described in Section 4.1.2. For the multimodal VQ-VAE model, the reconstructions of text embeddings, image embeddings and color categories need to be incorporated too. This results in a modified **reconstruction loss**, whose formulation is presented in Equation 4.20.

$$\begin{aligned}\mathcal{L}_{\text{reconstruction}} = ||b_i - \hat{b}_i||_2^2 - \alpha \sum_{j=0}^{nc-1} c_{i,j} \log \hat{c}_{i,j} + \gamma ||t_i - \hat{t}_i||_2^2 \\ + \delta ||v_i - \hat{v}_i||_2^2 - \epsilon \sum_{m=0}^{ncol-1} col_{i,m} \log \hat{col}_{i,m} \end{aligned} \tag{4.20}$$

In this equation:

- $nc$ is the total number of categories for GUI elements.
- $ncol$ is the total number of categories for colors.
- $||b_i - \hat{b}_i||_2^2$ is the MSE loss computed on the bounding box coordinates.
- $-\alpha \sum_{j=0}^{nc-1} c_{i,j} \log \hat{c}_{i,j}$ is the categorical Cross-Entropy loss computed using the predicted element categories.
- $\gamma ||t_i - \hat{t}_i||_2^2$: is the MSE loss computed on the text embeddings.
- $\delta ||v_i - \hat{v}_i||_2^2$ is the MSE loss computed on the image embeddings.
- $-\epsilon \sum_{m=0}^{ncol-1} col_{i,m} \log \hat{col}_{i,m}$ is the categorical Cross-Entropy loss computed using the predicted color categories.

$\alpha$, $\gamma$, $\delta$, and $\epsilon$ are scaling factors that balance the contribution of each term to the total reconstruction loss.

These additional loss terms encourage the model to learn representations that are effective for reconstructing not only the spatial and categorical information of GUI components but also their textual, visual, and color-related properties.

The **commitment loss** is unchanged from the formulation presented in Equation 4.13, ensuring that the continuous latent vectors produced by the encoder remain close to the discrete codebook vectors. This commitment loss applies to the entire latent representation, which now includes the additional information from the text embeddings, image embeddings, and color categories.

Therefore, the total loss function for the multimodal VQ-VAE model can be formulated as in Equation 4.21.

$$\mathcal{L} = \mathcal{L}_{reconstruction} + \beta \mathcal{L}_{commitment} \tag{4.21}$$

where $\mathcal{L}_{reconstruction}$ now includes the loss terms for text, images, and colors, and $\beta$ controls the relative importance of the commitment loss.

### 4.2.3 Implementation Details

According to the discussion in Section 3.2, the proposed multimodal VQ-VAE model for representation learning is trained and evaluated on FLUID only.

The implementation details for FLUID are the same reported in Section 4.1.3, except for the changes or additions highlighted in the following.

The additional embeddings and color categories require some preprocessing steps:

- For the color categories, the $k$-means algorithm from scikit-learn [70] is executed on all the training set colors to group them into $k = 22$ clusters. From these clusters, the centroids are extracted as the 22 color categories to use. Therefore, the colors of GUI elements are remapped to the nearest centroid and the result is one-hot encoded.

- As a consequence of the analysis in Figure 3.9, text elements with a length greater than 350 are truncated to 350 characters before being passed to BERT.

The multimodal VQ-VAE model is trained and tested on FLUID in two experiments. The first experiment uses the same configuration seen in Section 4.1.3. For the second experiment, $d$ is increased to 768, the number of multi attention heads is increased to 12, the dimension of the feedforward model is increased to 3,072, the number of layers is unchanged and is 12. The comparison is meant to assess whether a more complex model is able to achieve better results.

For the training parameters of the multimodal VQ-VAE model:

- The batch size for the training dataloader is reduced to 16.

- The patience for the *Early Stopping* algorithm is reduced to 16.

- The scaling factor $\gamma$ (see Equation 4.19) for the MSE loss computed on the text embeddings is set to 0.5.
- The scaling factor $\delta$ (see Equation 4.19) for the MSE loss computed on the image embeddings is set to 0.5.
- The scaling factor $\epsilon$ (see Equation 4.19) for the categorical Cross-Entropy loss computed on the color categories is set to 1 because of the increased number of color categories (i.e., 22) compared to the number of GUI element categories (i.e., 4), which may require further attention during optimization.

# Chapter 5

# Results

Following the methodology presented in Chapter 4, this Chapter reports the results of the Vector Quantized-Variational Autoencoder (VQ-VAE) model for representation learning of GUIs.

Specifically, Section 5.1 presents the evaluation metrics and the results of the VQ-VAE model for bounding box and category encoded representation learning.

Subsequently, Section 5.2 focuses on the integration of text embeddings, image embeddings and color categories to achieve multimodal representation learning, presenting the relevant metrics and the final results.

## 5.1 Bounding Box and Category Encoded Representation Learning

The evaluation details and the results of the VQ-VAE model for bounding box and category encoded representation learning are presented below.

The architecture, learning procedure and implementation details of the VQ-VAE model in this simple scenario are described in Sections 4.1.1, 4.1.2 and 4.1.3, respectively.

### 5.1.1 Evaluation Metrics

To evaluate the performance of the VQ-VAE model for bounding box and category encoded representation learning, two primary metrics are used: **accuracy** and **mean Intersection over Union** (mIoU). These metrics are chosen to assess the element category prediction and localization capabilities of the model.

Accuracy is a widely used metric to evaluate the proportion of correctly predicted instances among the total number of predictions, as described in Equation 5.1.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{5.1}$$

This metric is used to evaluate the capability of the model to correctly predict GUI element categories.

On the other hand, the mIoU metric is used to evaluate the quality of the reconstructed bounding boxes. It measures the overlap between the predicted bounding boxes and the ground truth, calculating the intersection over union (IoU) for each bounding box, and then averaging the results across all instances. The IoU for a single bounding box can be determined following the Equation 5.2.

$$\text{IoU} = \frac{|B_\text{p} \cap B_\text{gt}|}{|B_\text{p} \cup B_\text{gt}|} \tag{5.2}$$

where $B_\text{p}$ represents the predicted bounding box and $B_\text{gt}$ represents the ground-truth bounding box. $|B_\text{p} \cap B_\text{gt}|$ is the area of the intersection between the predicted and ground-truth boxes, while $|B_\text{p} \cup B_\text{gt}|$ is the area of their union. Consequently, the mean IoU (mIoU) is the average of the IoUs across all bounding boxes, as defined in Equation 5.3.

$$\text{mIoU} = \frac{1}{N} \sum_{i=1}^{N} \frac{|B_\text{p}^i \cap B_\text{gt}^i|}{|B_\text{p}^i \cup B_\text{gt}^i|} \tag{5.3}$$

where $N$ is the total number of bounding boxes.

The mIoU metric provides a comprehensive evaluation of both the position and the area of predicted bounding boxes by measuring the overlap between the predicted and ground-truth boxes. However, mIoU combines these aspects into a single score, making it difficult to isolate specific errors related to the position or size of the bounding box. To analyze the two aspects distinctly, two more metrics are used for the evaluation: the **position error** and the **area error**, derived from [71].

The position error, formally defined in Equation 5.4, measures the distance between the predicted position of the GUI element and the corresponding ground-truth position, normalized by the maximum possible distance the element could move within the interface.

$$\text{PosErr} = \frac{||(\hat{x}, \hat{y}) - (x, y)||_2}{\sqrt{(w_\text{UI} - \hat{w})^2 + (h_\text{UI} - \hat{h})^2}} \tag{5.4}$$

where:

- $(\hat{x}, \hat{y})$ are the predicted top-left coordinates of the bounding box.
- $(x, y)$ are the ground-truth top-left coordinates.

- $w_{\text{UI}}$ and $h_{\text{UI}}$ are the dimensions of the interface.
- $\hat{w}$ and $\hat{h}$ are the width and height of the predicted bounding box.

The position error scope is limited to the top-left coordinates of predicted and ground-truth bounding boxes. This prevents the position error from becoming overly biased toward an area metric, which would occur if the entire context of the bounding box, including the bottom-right coordinates, were considered in Equation 5.4.

Conversely, the area error measures the difference between the area of the predicted and ground-truth bounding boxes, capturing the accuracy of the bounding box in terms of size. It is computed as the difference between the predicted and ground-truth area normalized by the maximum value between the two, as shown in Equation 5.5.

$$\text{AreaErr} = \frac{|\hat{w} \cdot \hat{h} - w \cdot h|}{\max(\hat{w} \cdot \hat{h}, w \cdot h)} \tag{5.5}$$

where:

- $\hat{w} \cdot \hat{h}$ is the area of the predicted bounding box.
- $w \cdot h$ is the area of the ground-truth bounding box.

### 5.1.2 Results and Discussion

The proposed VQ-VAE model is evaluated, considering all the metrics described in 5.1.1, on both the Rico [4] dataset and FLUID (see Chapter 3), to assess its overall performance in bounding box reconstruction and category prediction.

Table 5.1 reports the mIoU, the accuracy, the position error and the area error computed during the VQ-VAE model evaluation on the test set, on both the aforementioned datasets.

| **Dataset** | **mIoU ↑** | **Acc ↑** | **PosErr ↓** | **AreaErr ↓** |
|---|---|---|---|---|
| Rico | $\mathbf{0.59 \pm 0.18}$ | $99.61 \pm 2.21$ | $0.03 \pm 0.04$ | $\mathbf{0.21 \pm 0.11}$ |
| FLUID | $0.37 \pm 0.10$ | $\mathbf{99.88 \pm 0.71}$ | $\mathbf{0.02 \pm 0.01}$ | $0.31 \pm 0.08$ |

**Table 5.1:** VQ-VAE evaluation results for bounding box reconstruction and category prediction on Rico and FLUID. **Bold** values indicate the overall best results

The discussion in Section 3.2 explained the motivations behind the Rico dataset unsuitability for multimodal GUI representation learning. Instead, in the simple

scenario of bounding box reconstruction and category prediction, there are no downsides to using it. Therefore, Table 5.1 reports the results of the proposed VQ-VAE model also for Rico, in the simple scenario of bounding box reconstruction and category prediction. Despite not being the focus of this work, as the objective is to develop an AI model able to learn a meaningful representation for different kinds of GUI elements, including images, these preliminary results on the Rico dataset can prove to be a reliable benchmark for the VQ-VAE learning abilities against both recent and future works. This is particularly relevant if we consider that most of the works presented in Section 2.2.2 only provide results in terms of downstream tasks performance and use representation learning as a means to achieve a specific outcome, rather than evaluating the overall representation capabilities of the latent space. Jing et al. [29] provide a benchmark for the representation learning scenario, but the metrics are different from the ones used in this work. Specifically, the authors in [29] used a max IoU metric, which most likely only gives overoptimistic results compared to the mIoU defined in Equation 5.3, since it searches for the best possible permutation of elements to compute the maximum average IoU. Moreover, even considering it as an overoptimistic benchmark to compare with, [29] only achieves a max IoU of 0.55, whereas **the VQ-VAE model proposed in this work achieves a mean IoU of 0.59**, with possibly even higher IoU results on some specific GUI layouts.

Comparing the results on both datasets, **the VQ-VAE achieves an accuracy greater than 99% for GUI element prediction**. The VQ-VAE is able to operate more uniformly on FLUID than Rico, as it can be seen from the respective standard deviations for accuracy, that are 0.71 and 2.21. This behavior can be explained by the differences in the two datasets: while FLUID has only 4 categories (i.e., `IMAGE`, `TEXT`, `SOLID`, `SVG`) for GUI elements, the 25 categories available in Rico are making the training process more difficult. Instead, for bounding box reconstruction, the mIoU computed for FLUID is significantly lower than the one computed for Rico. These results can be further analyzed thanks to the additional position error and area error metrics, defined in Equations 5.4 and 5.5, respectively. Their inspection in Table 5.1 suggests that **the VQ-VAE model is able to place the top-left corners of the bounding boxes for both datasets at approximately correct locations**, given the low position error. Conversely, **the area error is moderately high, especially for FLUID, and ultimately contributes to the reduction in mIoU experienced on such dataset**. Therefore, from these results it can be inferred that the position of bounding boxes in reconstructed GUIs will most likely be approximately correct, while their area will probably be less accurate.

Figures 5.1 and 5.2 offer a visualization for the average area and position errors as functions of the number of existing elements within GUIs, for Rico and FLUID, respectively.

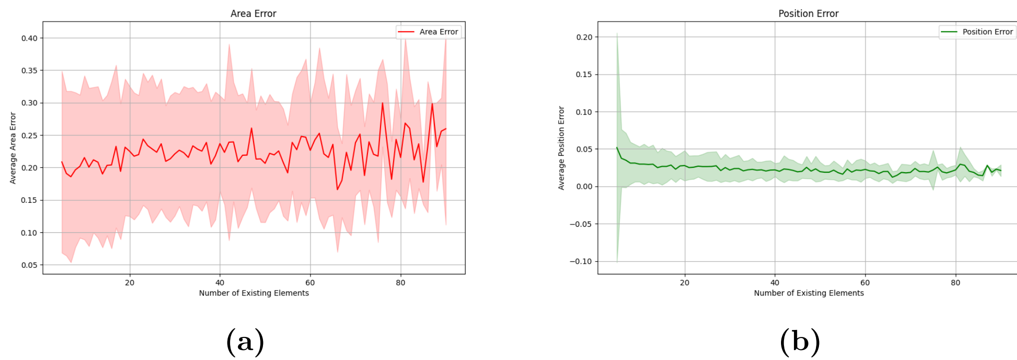



**Figure 5.1:** Average area and position errors for Rico. (a): Average area error. (b): Average position error

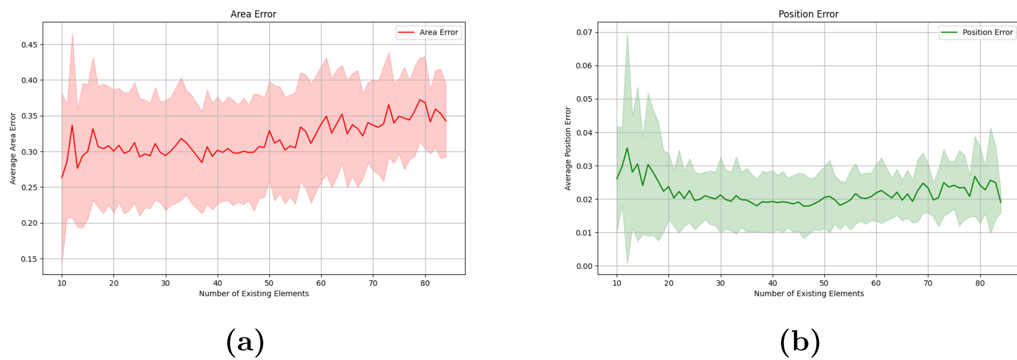



**Figure 5.2:** Average area and position errors for FLUID. (a): Average area error. (b): Average position error

The area errors shown in Figures 5.1a and 5.2a suggest a positive correlation, for both datasets, with the number of elements within GUIs. **The VQ-VAE model better addresses the reconstruction of GUIs with few elements**.

Figures 5.3, 5.4, 5.5 and 5.6 show some visual examples for the VQ-VAE bounding box reconstructions and category predictions on FLUID.

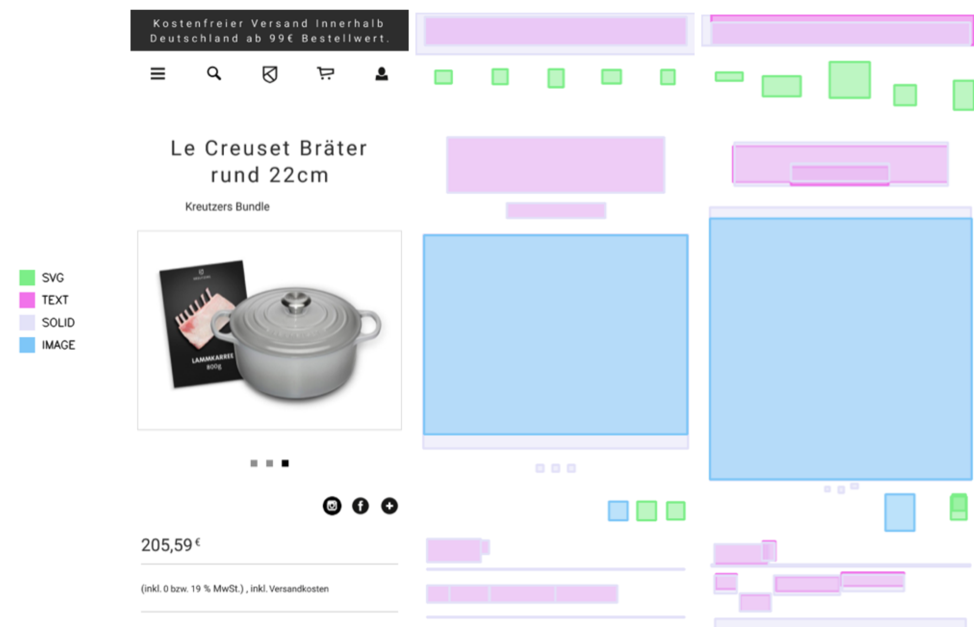



**Figure 5.3:** Example #1 of GUI from FLUID and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

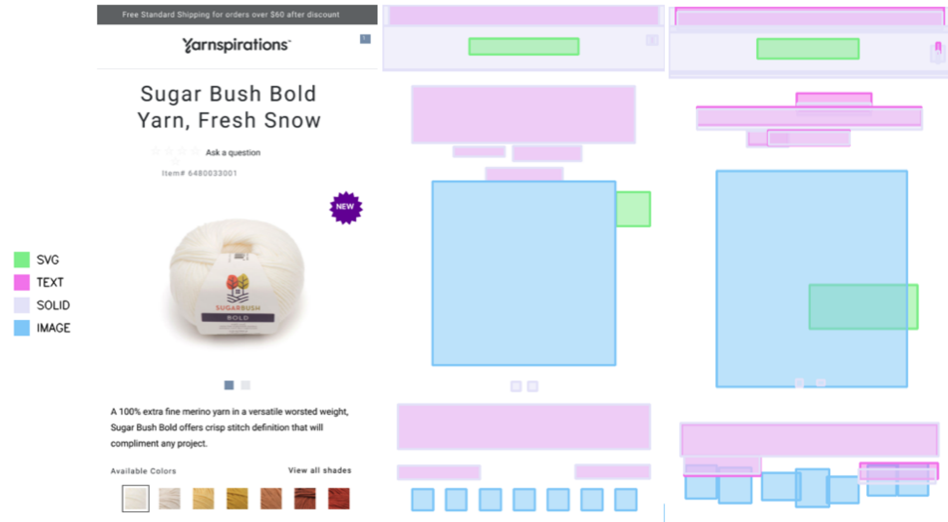



**Figure 5.4:** Example #2 of GUI from FLUID and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

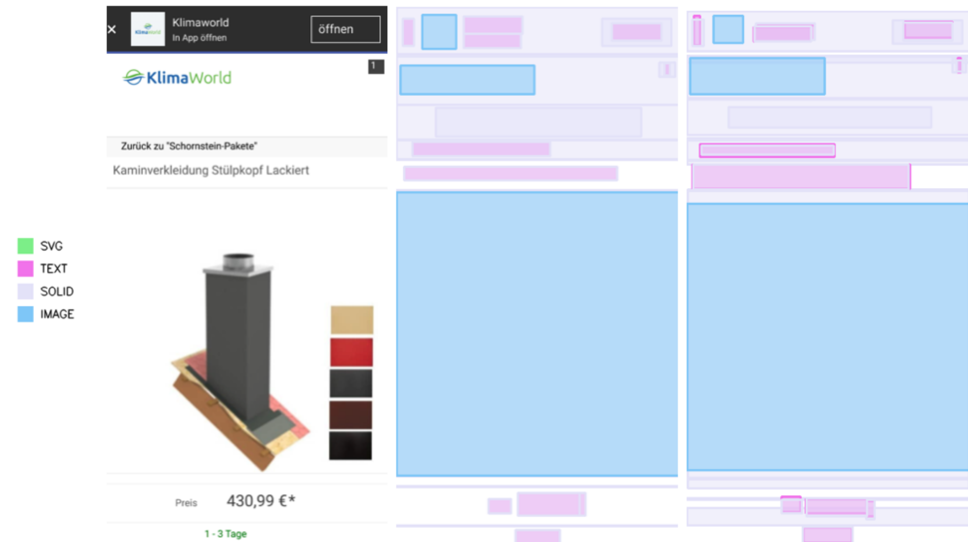



**Figure 5.5:** Example #3 of GUI from FLUID and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

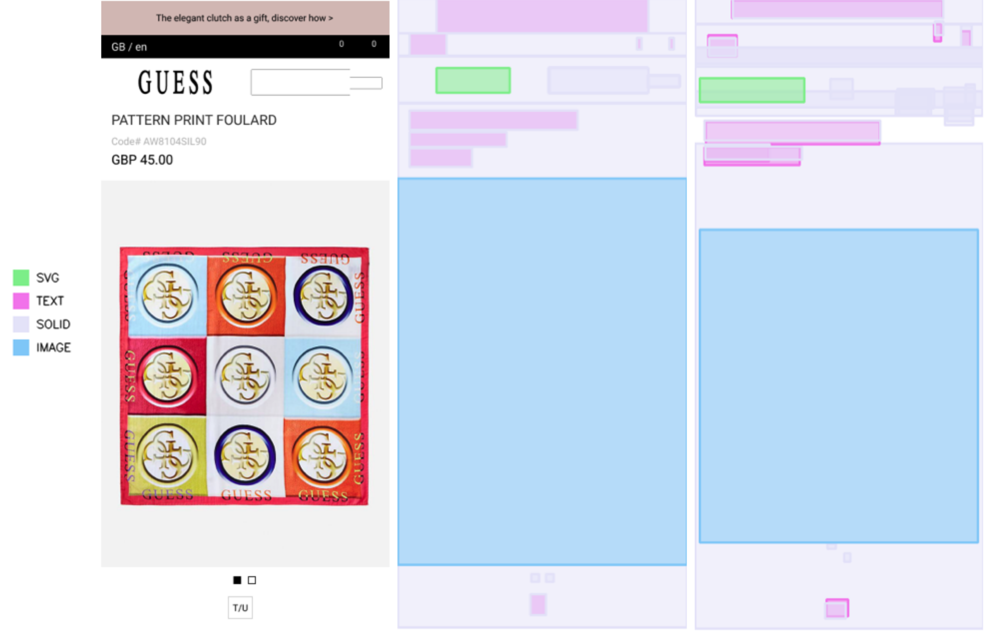



**Figure 5.6:** Example #4 of GUI from FLUID and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

The categories are accurately predicted in all four examples. For the bounding boxes, the reconstruction in Figure 5.5 is mostly correct, except for the GUI elements on the bottom part that originally had a low height and now appear as larger. The reconstructions in Figures 5.3, 5.4 and 5.5 show some errors, especially for small GUI elements, that seem to be the most problematic ones. Specifically, the reconstructions in Figures 5.3, 5.4 and 5.6 show that the squared elements related to the carousel are slightly misplaced. However, **given their reduced size, even a slight misalignment can cause the intersection between the original bounding box and the reconstructed bounding box to become low, possibly even zero**.

**This may be the cause for the reduced mIoU on the FLUID dataset**, which may have a high number of small GUI elements whose impact on the results may be non-negligible.

Similarly, Figures 5.7, 5.8, 5.9 and 5.10 show some visual examples for the VQ-VAE bounding box reconstructions and category predictions on Rico.

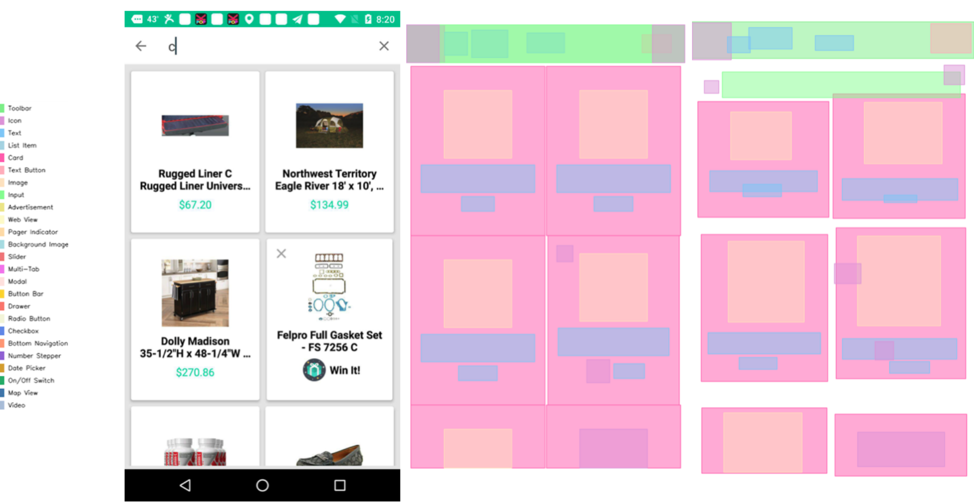



**Figure 5.7:** Example #1 of GUI from Rico and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

The categories are accurately predicted. For bounding boxes reconstruction, the examples show good results, with an almost perfect reconstruction for the GUI shown in Figure 5.9. Notably, **there are some GUI elements that appear small in Rico GUIs too**. However, considering that the GUI size in Rico is 1,440 × 2,560 pixels, compared to 375 × 1,400 pixels in FLUID, their size is most likely greater, in absolute terms, with respect to the small elements seen in FLUID

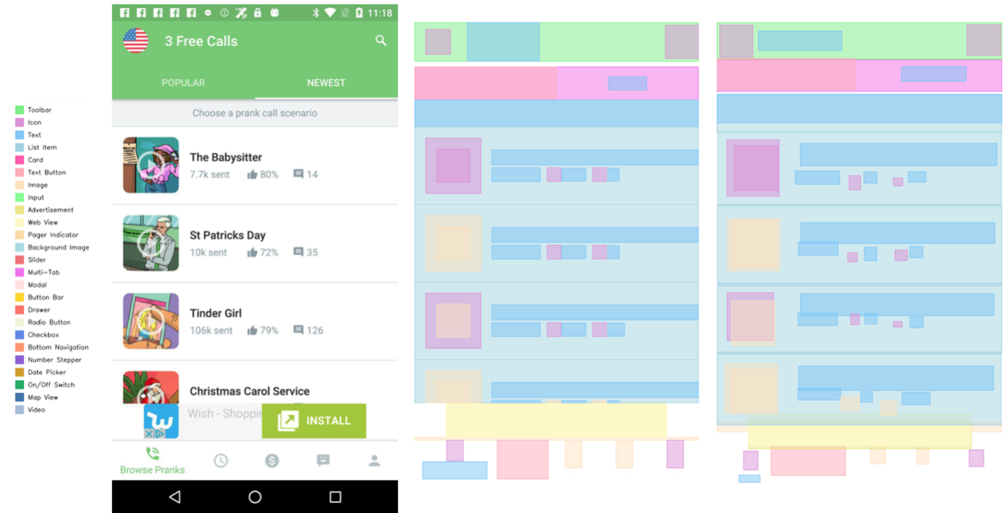



**Figure 5.8:** Example #2 of GUI from Rico and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

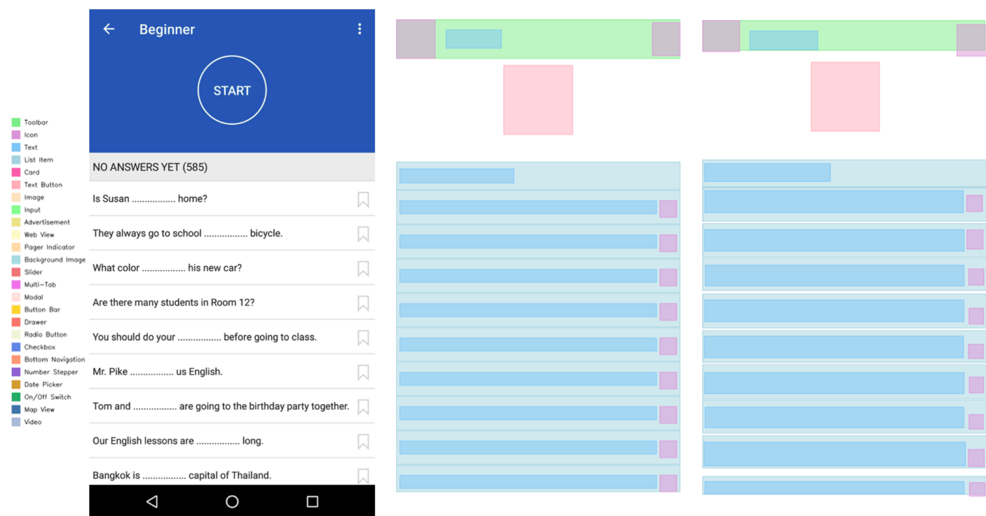



**Figure 5.9:** Example #3 of GUI from Rico and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

GUIs. Therefore, **their effect on the mIoU could be lower than for FLUID, allowing to achieve better mIoU results**.

The distribution of the areas for FLUID and Rico bounding boxes of the test

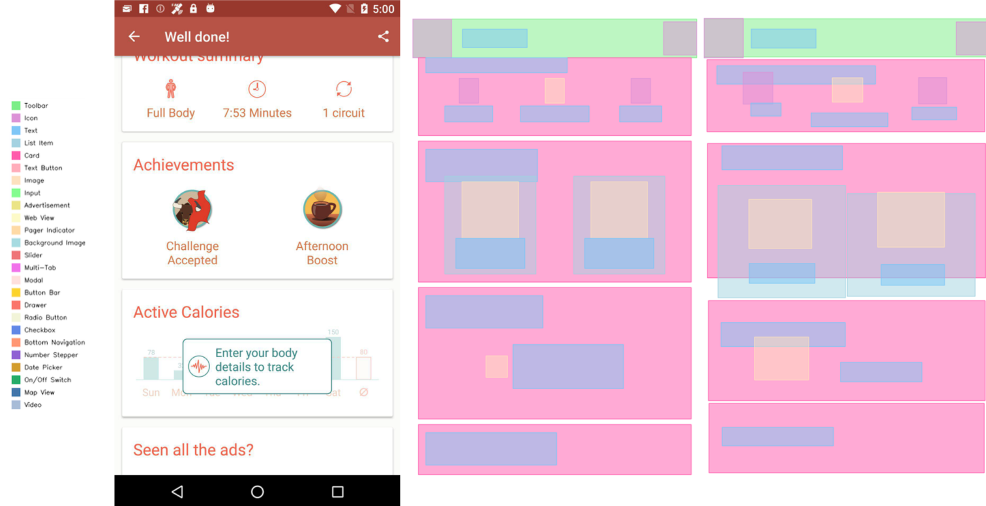
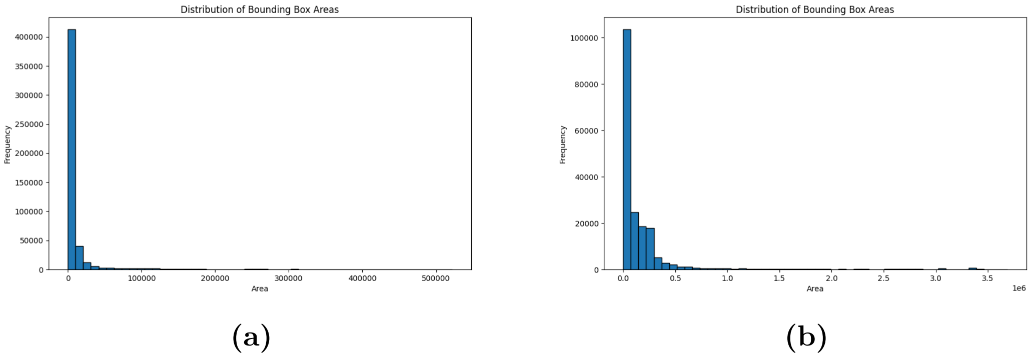



**Figure 5.10:** Example #4 of GUI from Rico and its VQ-VAE reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE reconstructed bounding boxes. Colors represent the GUI element categories

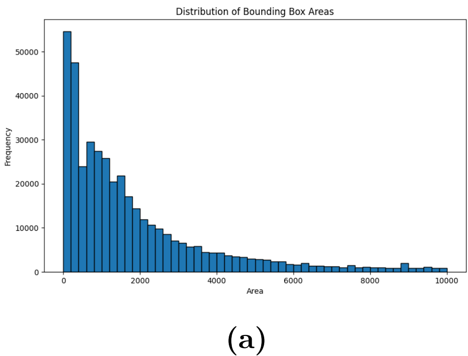
set is shown in Figure 5.11, to evaluate whether the increased presence of small GUI elements in FLUID than Rico could indeed be influencing the results.



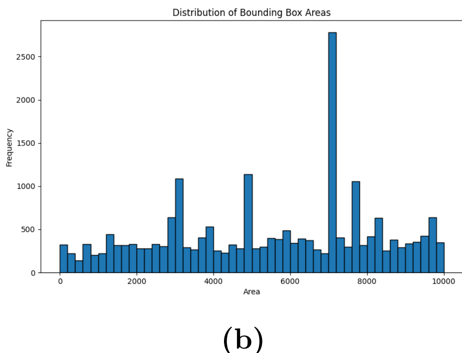

**(a)** **(b)**

**Figure 5.11:** Distribution of the areas for FLUID and Rico bounding boxes of the test set. (a): FLUID. (b): Rico

The distribution for FLUID in Figure 5.11a shows a higher frequency for small area values, compared with Rico. To better analyze these results, the analysis is restricted to the [0, 10,000] interval for the $x$ axis and the results are shown in Figure 5.12.




**(a)**




**(b)**

**Figure 5.12:** Distribution of the areas for FLUID and Rico bounding boxes of the test set, on a restricted interval. (a): FLUID. (b): Rico

The distributions confirm that **for FLUID there is a higher number of small GUI elements compared with Rico, for the test sets**.

**An additional evaluation is carried out on both FLUID and Rico to empirically assess the results if small elements were to be ignored for the computation of the mIoU**. Specifically, an area threshold $th$ is arbitrarily chosen to ignore GUI elements whose area is lower than this value. Given the distributions presented in Figure 5.12, for this evaluation $th = 2{,}000$. The results are shown in Table 5.2.

| **Dataset** | **mIoU** $\uparrow$ |
|---|---|
| Rico | $0.59 \pm 0.18$ |
| Rico ($th = 2{,}000$) | $\mathbf{0.60 \pm 0.18}$ |
| FLUID | $0.37 \pm 0.10$ |
| FLUID ($th = 2{,}000$) | $\mathbf{0.49 \pm 0.12}$ |

**Table 5.2:** VQ-VAE mIoU evaluation results for bounding box reconstruction on Rico and FLUID, including the test without small GUI elements. **Bold** values indicate the overall best results

The results on FLUID are still not on par with the ones on Rico. However, the mIoU improvement for FLUID is evidently more pronounced than the improvement experienced for Rico, **confirming that the lower mIoU of FLUID is partially caused by the high presence of small GUI elements in the test set**. The small elements are naturally harder to target in reconstruction, and future work may explore alternative strategies and training optimizations to more accurately reconstruct them.

## 5.2 Multimodal Encoded Representation Learning

The evaluation details and the results of the VQ-VAE for multimodal encoded representation learning of GUIs, with the inclusion of text embeddings, image embeddings and color categories, are presented below.

The architecture is the same one used for the VQ-VAE in the simple scenario with only bounding boxes and element categories, described in Section 4.1.1. The learning procedure is adjusted to incorporate additional terms in the reconstruction loss, as presented in Section 4.2.2. The implementation details for the VQ-VAE in this complex scenario are reported in Section 4.2.3.

The multimodal scenario only involves FLUID, for the reasons documented in Section 3.2.

Despite the results in Section 5.1 pointing to low bounding box reconstruction performance for small elements of GUIs, no adjustments are made to address this issue. Specifically, **small GUI elements are not removed from the training set to provide the model with a comprehensive, representative of real-world applications, set of GUIs**. Similarly, **they are not removed from the test set, as this would provide overoptimistic results** that do not consider a realistic GUI layout may contain small elements.

So, the idea is to retain the bounding box reconstruction training and evaluation procedures as the ones used for the simple scenario, effectively testing the model under stressful conditions.

### 5.2.1 Evaluation Metrics

**Cosine similarity** has become the standard metric for evaluating the similarity between two embeddings in vector space. Introduced initially in the context of information retrieval [72], it has since been widely adopted in modern embedding-based models [22, 73, 69] for its effectiveness in capturing semantic similarity through the measurement of directional alignment rather than magnitude. In this work, embeddings are designed to encode semantic properties, therefore cosine similarity is well-suited as it provides a robust, magnitude-invariant measure of similarity that highlights the alignment of the learned representations.

Given two embeddings, $x$ and $y$, the cosine similarity is defined as in Equation 5.6.

$$\text{Cosine Similarity} = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \tag{5.6}$$

The cosine similarity varies between -1 and 1, with 1 indicating perfect similarity.

This metric is suitable for evaluating the similarity of both image and text embeddings, which are introduced in the multimodal context.

For the color categories predictions, the **accuracy**, previously reported in Equation 5.1, can be used in the same way as for GUI element categories.

### 5.2.2 Results and Discussion

As reported in Section 4.2.3, **two different configurations of the VQ-VAE are tested for the multimodal representation learning of GUIs**. The first one adheres with the VQ-VAE used for the simple scenario and is called VQ-VAE512, reflecting the fact that $d = 512$. The second one employs a more complex transformer and is called VQ-VAE768, since $d = 768$ in this configuration.

The two VQ-VAE configurations are evaluated on FLUID considering all the metrics described in Section 5.1.1, with the addition of the metrics in Section 5.2.1, specifically tailored to image and text embedding similarity, and color category prediction accuracy.

Table 5.3 reports the result computed on FLUID during the VQ-VAE model evaluation on the test set.

| **Model** | **mIoU** $\uparrow$ | **Acc** $\uparrow$ | **PosErr** $\downarrow$ | **AreaErr** $\downarrow$ | $\mathbf{Acc_{col}}$ $\uparrow$ | $\mathbf{Cos_{txt}}$ $\uparrow$ | $\mathbf{Cos_{img}}$ $\uparrow$ |
|---|---|---|---|---|---|---|---|
| VQ-VAE512 | $0.10 \pm 0.05$ | $98.63 \pm 2.81$ | $0.08 \pm 0.03$ | $0.51 \pm 0.11$ | $\mathbf{99.63 \pm 2.61}$ | $0.84 \pm 0.06$ | $0.81 \pm 0.09$ |
| VQ-VAE768 | $\mathbf{0.11 \pm 0.05}$ | $\mathbf{98.90 \pm 2.59}$ | $\mathbf{0.07 \pm 0.03}$ | $0.51 \pm 0.11$ | $99.59 \pm 2.26$ | $0.84 \pm 0.06$ | $0.81 \pm 0.09$ |

**Table 5.3:** VQ-VAE evaluation results on FLUID for multimodal representation learning. **Bold** values indicate the overall best results

The VQ-VAE768 achieves only marginally better results than the VQ-VAE512 configuration in terms of mIoU, accuracy for GUI element categories and position error. Despite the higher mean for the accuracy of color category prediction offered by the VQ-VAE512, the VQ-VAE768 configuration achieves similar results with a reduced standard deviation, pointing to more uniform performance on this task. Overall, **for both models the accuracy for GUI element category prediction and the one for color category prediction highlight that the VQ-VAE excels in these tasks**. The cosine similarity results underscore that **both models are able to reconstruct the text and image embeddings with a good degree of fidelity**. **The mIoU and area error still indicate low performance**, even lower than the results in the simple scenario (see Section 5.1.2). This behavior may be attributed to:

- The mIoU issues with small GUI elements, highlighted in Section 5.1.2.
- The more complex training optimization procedure, which now involves not only reconstructing bounding boxes and predicting GUI element categories,

but also predicting color categories and learning a representation for text and images. Moreover, the performance can also be highly dependent on the chosen scaling factors $\alpha$, $\beta$, $\gamma$, $\delta$ and $\epsilon$ (see Equations 4.19 and 4.21) for the loss terms, that are hyperparameters.

Figures 5.13 and 5.14 show the average area and position errors on FLUID as functions of the number of existing elements withing GUIs, for VQ-VAE512 and VQ-VAE768, respectively, as previously done for the simple scenario (see Figure 5.2).

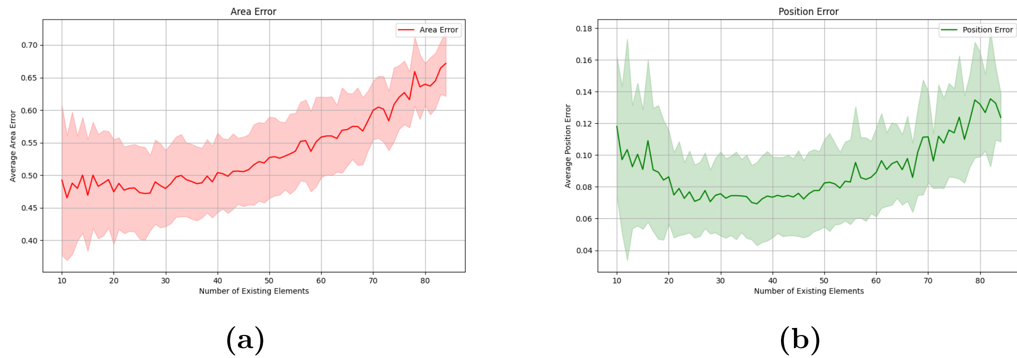



**Figure 5.13:** VQ-VAE512 average area and position errors for FLUID. (a): Average area error. (b): Average position error

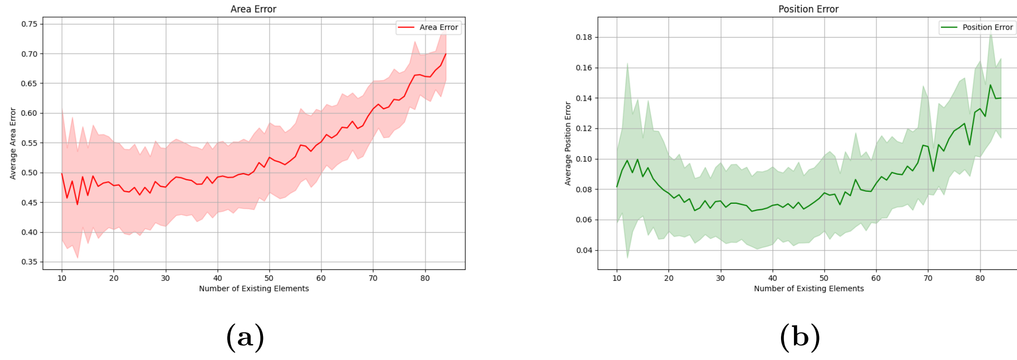



**Figure 5.14:** VQ-VAE768 average area and position errors for FLUID. (a): Average area error. (b): Average position error

The results of the two configurations are very similar. Compared to the simple scenario in Figure 5.2, **they confirm the higher complexity of multimodal representation learning**. In fact, the area error now shows even worse performance when the number of elements within GUIs increases. For the position error,

which in the simple scenario (Figure 5.2b) did not exhibit this behavior, now there is also a positive correlation with the number of GUI elements, indicating that **the multimodal VQ-VAEs struggle with the placement of GUI elements when there are many of them**.

Figures 5.15, 5.16, 5.17 and 5.18 show the visual results for the VQ-VAE512 bounding box reconstruction and category prediction on the same FLUID GUIs presented in Figures 5.3, 5.4, 5.5 and 5.6.

**The GUIs reconstructed using the VQ-VAE512 model appear more confused**, further underscoring the complexity of the bounding box reconstruction task in the multimodal scenario. Although big elements are approximately correctly reconstructed, the smaller ones continue to be misplaced and/or wrongly sized.

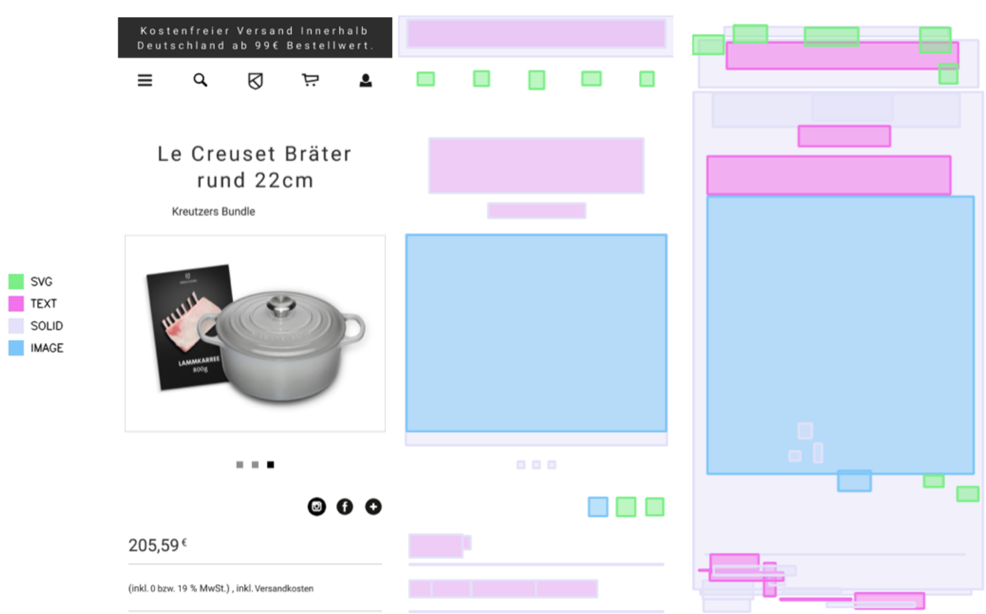



**Figure 5.15:** Example #1 of GUI from FLUID and its VQ-VAE512 reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE512 reconstructed bounding boxes. Colors represent the GUI element categories

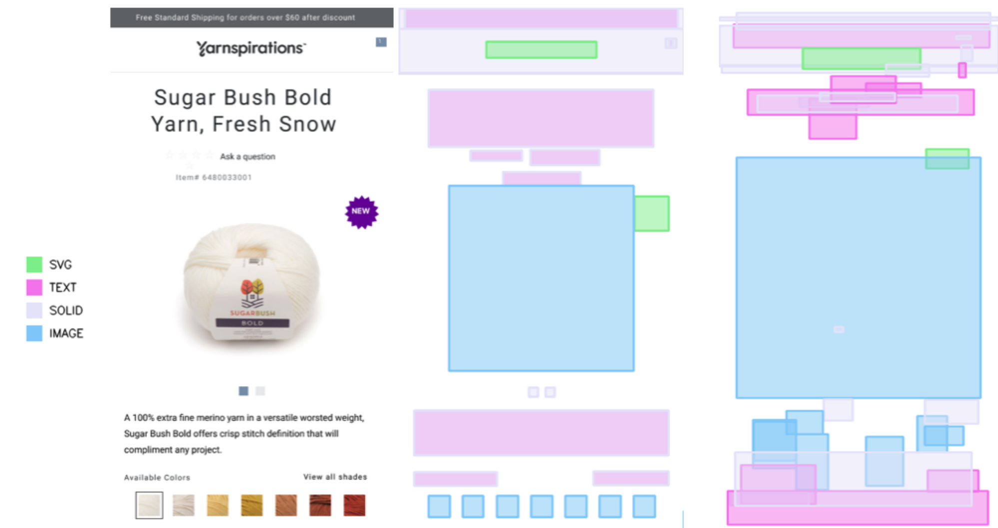



**Figure 5.16:** Example #2 of GUI from FLUID and its VQ-VAE512 reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE512 reconstructed bounding boxes. Colors represent the GUI element categories

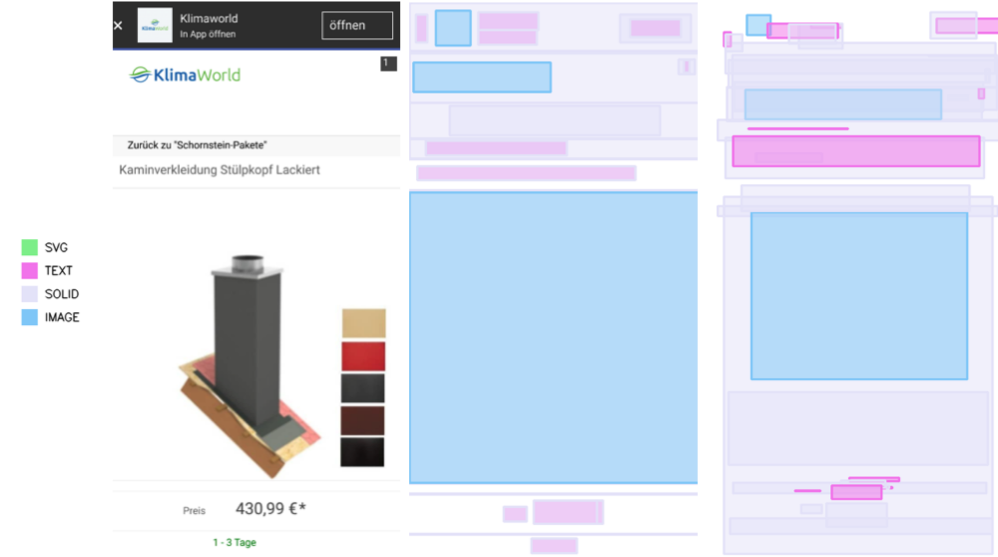



**Figure 5.17:** Example #3 of GUI from FLUID and its VQ-VAE512 reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE512 reconstructed bounding boxes. Colors represent the GUI element categories

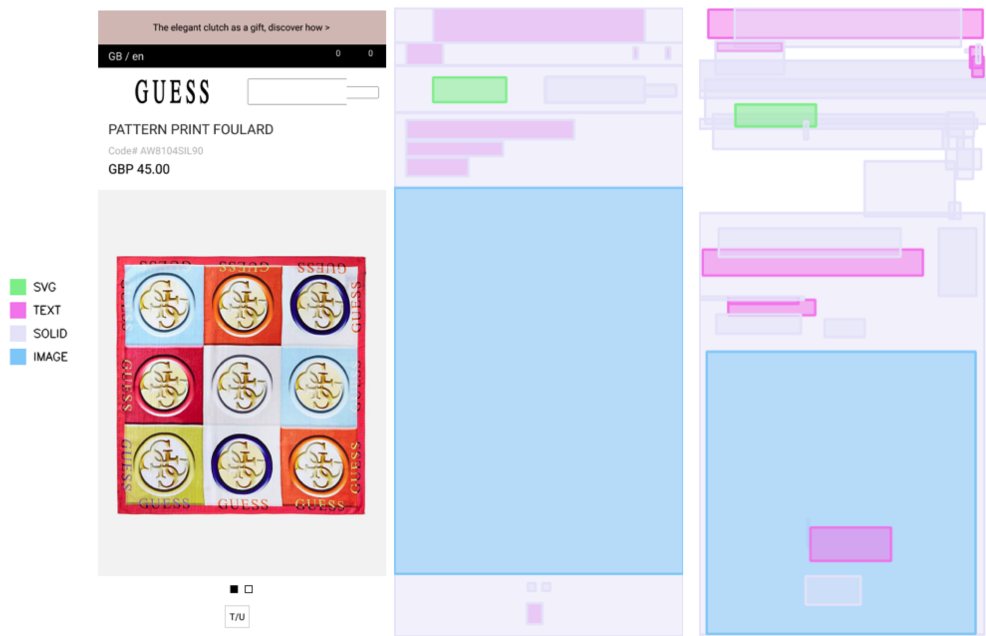



**Figure 5.18:** Example #4 of GUI from FLUID and its VQ-VAE512 reconstruction. Left: original GUI. Center: original GUI bounding boxes. Right: VQ-VAE512 reconstructed bounding boxes. Colors represent the GUI element categories

The same GUIs and their reconstructions are shown in Figures 5.19, 5.20, 5.21 and 5.22 for the VQ-VAE768 bounding box reconstruction and category prediction. The visualization includes the VQ-VAE512 reconstruction to make the comparison easier.

Overall, there are cases where the VQ-VAE768 seems to provide better GUI element placement (see Figure 5.20), and others where the result is even less accurate than the VQ-VAE512 reconstruction (see Figure 5.22). **The more powerful transformer used for the VQ-VAE768 does not provide the expected benefits for the bounding box reconstruction task**.

Figures 5.23 and 5.24 show the results for the image reconstruction task using the VQ-VAE512. To extract the reconstructed images, the following steps are performed:

1. First, images from the training set are passed to CLIP (Contrastive Language-Image Pretraining) [69] to determine and store a mapping between training images and their embeddings.

2. Then, the GUIs from the test set are passed to the VQ-VAE512, to obtain their reconstructed versions. For GUIs that contain images, these include the reconstructed image embeddings.

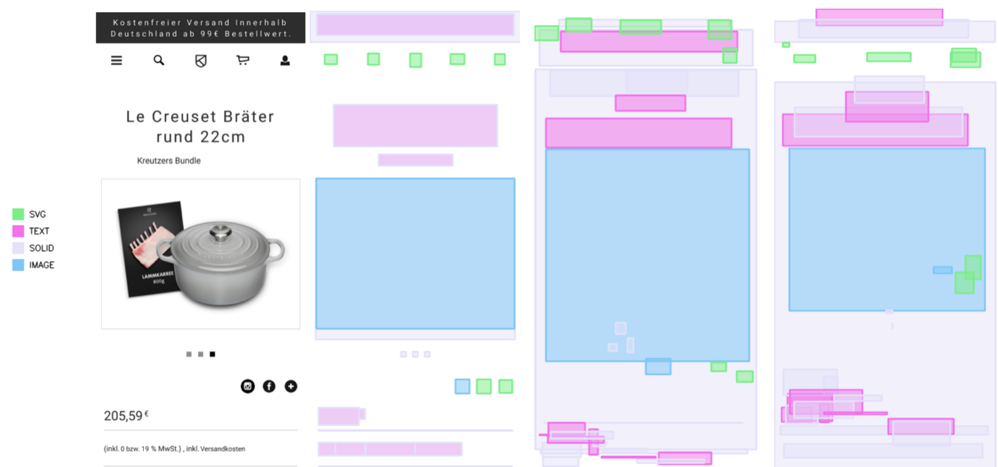



**Figure 5.19:** Example #1 of GUI from FLUID and its VQ-VAE768 reconstruction. Starting from the left: original GUI; original GUI bounding boxes; VQ-VAE512 reconstructed bounding boxes; VQ-VAE768 reconstructed bounding boxes. Colors represent the GUI element categories

**Figure 5.20:** Example #2 of GUI from FLUID and its VQ-VAE768 reconstruction. Starting from the left: original GUI; original GUI bounding boxes; VQ-VAE512 reconstructed bounding boxes; VQ-VAE768 reconstructed bounding boxes. Colors represent the GUI element categories

3. The reconstructed image embeddings are compared with the training set embeddings using the cosine similarity metric, to extract the top 3 training images whose embeddings are most similar to the one reconstructed at test time.

Figure 5.23 highlights that there are some images whose embeddings are most

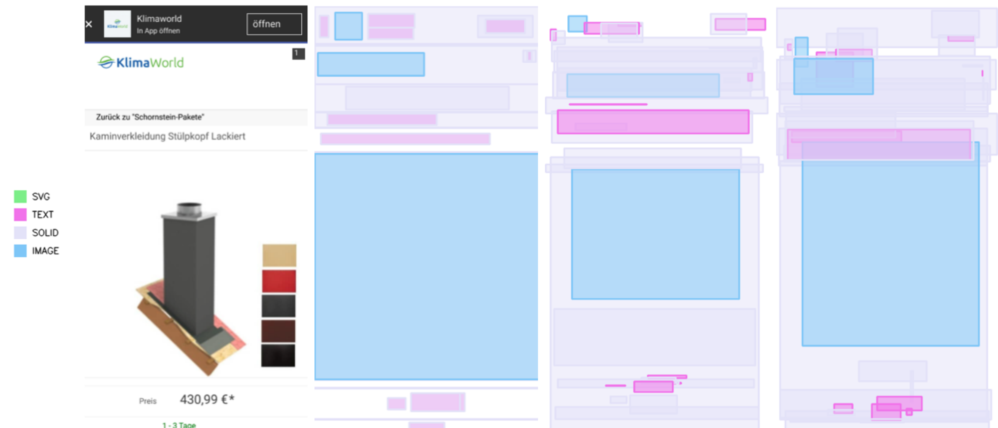



**Figure 5.21:** Example #3 of GUI from FLUID and its VQ-VAE768 reconstruction. Starting from the left: original GUI; original GUI bounding boxes; VQ-VAE512 reconstructed bounding boxes; VQ-VAE768 reconstructed bounding boxes. Colors represent the GUI element categories

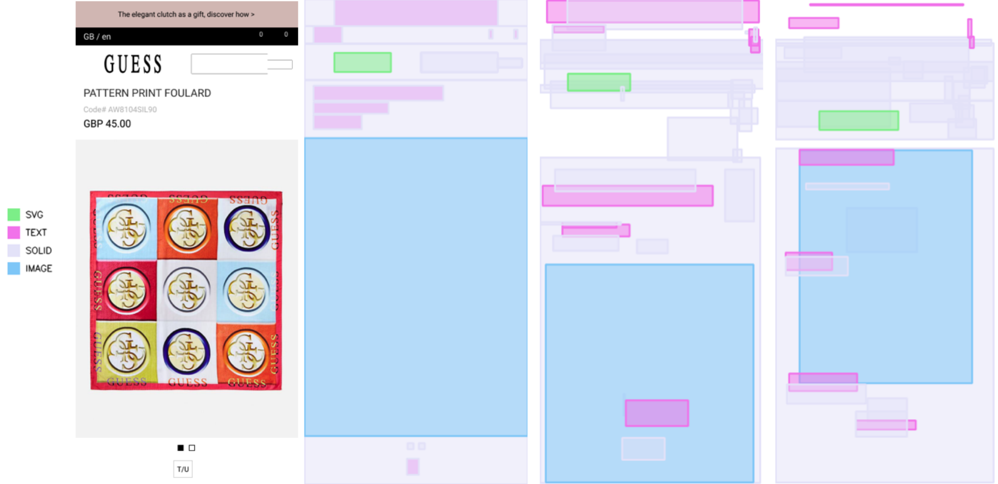



**Figure 5.22:** Example #4 of GUI from FLUID and its VQ-VAE768 reconstruction. Starting from the left: original GUI; original GUI bounding boxes; VQ-VAE512 reconstructed bounding boxes; VQ-VAE768 reconstructed bounding boxes. Colors represent the GUI element categories

similar to the ones of training images representing the same object. Conversely, Figure 5.24 shows some images whose embeddings are most similar to the ones of training images representing completely different objects. Combined with the high cosine similarity score in Table 5.3, the VQ-VAE512 generally appears to perform

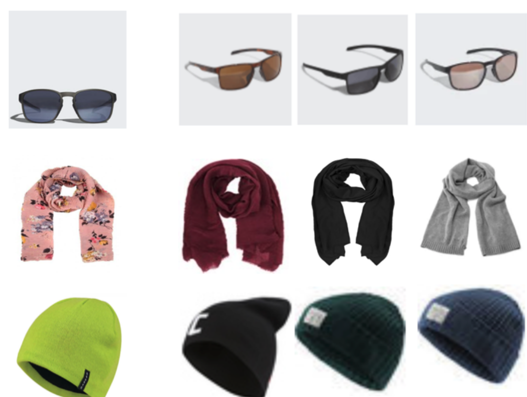


**Figure 5.23:** Examples of images correctly reconstructed by the VQ-VAE512. Left: Original image. Right: Top 3 training set images whose embeddings are most similar to the reconstructed image embedding

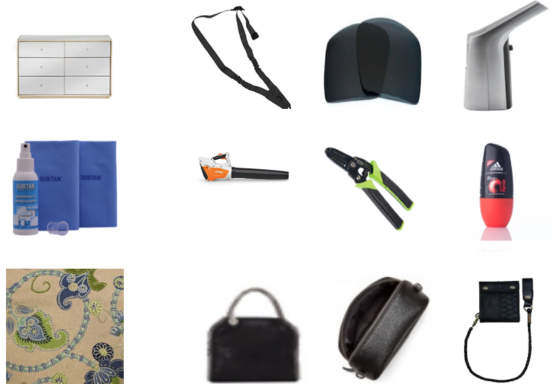


**Figure 5.24:** Examples of images wrongly reconstructed by the VQ-VAE512. Left: Original image. Right: Top 3 training set images whose embeddings are most similar to the reconstructed image embedding

well for the image reconstruction task, except for some edge cases where it fails.

The same visualization for the image reconstruction task using the VQ-VAE768 is shown in Figures 5.25 and 5.26.

This procedure can be applied to SVGs too. The results for the reconstruction of SVGs using the VQ-VAE512 are shown in Figures 5.27 and 5.28.

The same is done for the VQ-VAE768, and the results for the reconstruction of SVGs are shown in Figures 5.29 and 5.30.

These operations can be used to evaluate the results of the text reconstruction task, with the only difference being that, instead of CLIP, BERT (Bidirectional

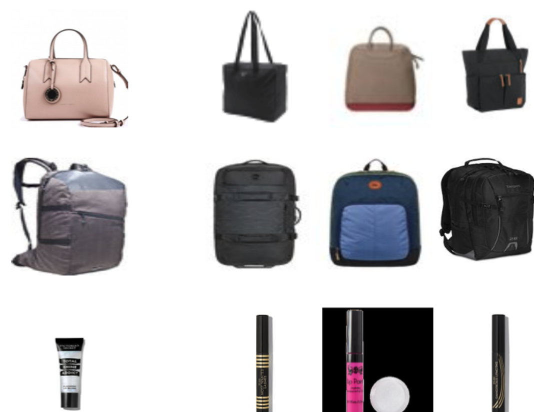


**Figure 5.25:** Examples of images correctly reconstructed by the VQ-VAE768. Left: Original image. Right: Top 3 training set images whose embeddings are most similar to the reconstructed image embedding

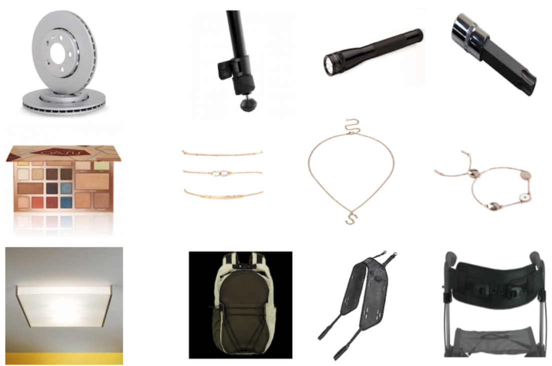


**Figure 5.26:** Examples of images wrongly reconstructed by the VQ-VAE768. Left: Original image. Right: Top 3 training set images whose embeddings are most similar to the reconstructed image embedding

Encoder Representations from Transformers) [61] is used to obtain the embeddings for the training texts. The results for both the VQ-VAE512 and VQ-VAE768 models are shown in Tables 5.4 and 5.5, limited to the top 1 most similar text.

The text samples that are correctly reconstructed are mostly related to GUI elements that can be found more frequently within interfaces. Instead, the cases where the VQ-VAEs fail are mostly texts that are related to specific products or objects.

**Figure 5.27:** Examples of SVGs correctly reconstructed by the VQ-VAE512. Left: Original SVG. Right: Top 3 training set SVGs whose embeddings are most similar to the reconstructed SVG embedding

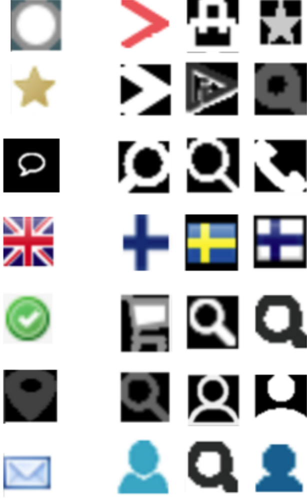


**Figure 5.28:** Examples of SVGs wrongly reconstructed by the VQ-VAE512. Left: Original SVG. Right: Top 3 training set SVGs whose embeddings are most similar to the reconstructed SVG embedding

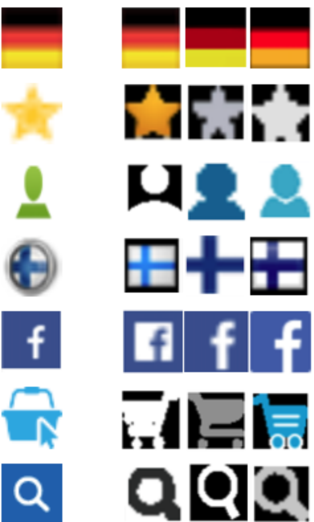


**Figure 5.29:** Examples of SVGs correctly reconstructed by the VQ-VAE768. Left: Original SVG. Right: Top 3 training set SVGs whose embeddings are most similar to the reconstructed SVG embedding

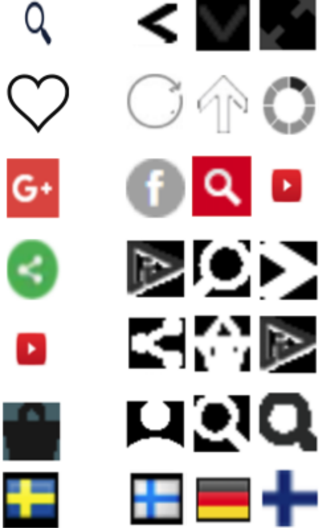


**Figure 5.30:** Examples of SVGs wrongly reconstructed by the VQ-VAE768. Left: Original SVG. Right: Top 3 training set SVGs whose embeddings are most similar to the reconstructed SVG embedding

| Model | Original text | Reconstructed text |
|---|---|---|
| VQ-VAE512 | Home | Home |
| | Shop Now | SHOP NOW |
| | 2 reviews | 3 Reviews |
| | Size: | Size: |
| | Free US shipping over $40 + free returns. | FREE SHIPPING ON ORDERS $50+. FREE RETURNS. |
| | Favorite | Favorite |
| VQ-VAE768 | Menü | Menu |
| | In den Warenkorb | In den Warenkorb |
| | Artikel-Nr.: | Artikel-Nr.: |
| | Color | Colour |
| | Write a Review | Write a Review |
| | In Stock | In Stock |

**Table 5.4:** Examples of text correctly reconstructed by the VQ-VAE512 and VQ-VAE768

| Model | Original text | Reconstructed text |
|---|---|---|
| VQ-VAE512 | Dorala Artwork Scarf | Chain Detail Bag |
| | ASOS DESIGN half moon marble clutch bag | Ryedale Peony - Bonbon 24" Wheel Girls' Bike |
| | Soda | Decor |
| | Shades of purple seed bead bracelet | Christmas Sequins Ball Decoration Santa Earrings - Gold |
| | Coffee Supplies | Car Charms |
| VQ-VAE768 | Herschel Supply Co. Little America Black 25L Backpack | Ice scratcher kit simple |
| | Filing\|shelving | Coffee supllies |
| | Toilets, Sinks, Faucets & Plumbing Supplies | Vacuum Cleaners |
| | Bottle – Measuring - 8 Ounce - Double Neck - Each | DuoClean Cordless Upright Vacuum Cleaner with Powered Lift-Away IC160UK |
| | Telescope Handsfree | Nordic Ware Rolled Omelet |

**Table 5.5:** Examples of text wrongly reconstructed by the VQ-VAE512 and VQ-VAE768

# Chapter 6

# Conclusion

This research aimed to facilitate both the development and the deployment of new Artificial Intelligence (AI) models for downstream real-world applications in GUI design.

First, a literature review is presented, covering the traditional GUI design process, how Machine Learning (ML) can fit into this context to enhance the designers' workflow, and finally presenting related works for GUI representation learning.

Then, a new dataset is proposed, called FLUID, Figma Layout User Interface Dataset. The dataset is built, starting from WebColor, through iterative analysis and refinement. FLUID is built for compatibility with Figma, enabling its files to be opened and edited directly within the Figma application. Therefore, using FLUID, the internal GUI representation used by Figma is available for training new AI models. The output of these models will remain fully compliant with the Figma representation, ensuring they can be seamlessly deployed within a real-world, widely-used GUI design framework.

To support the development of new AI models for downstream tasks, a Vector Quantized-Variational Autoencoder (VQ-VAE) architecture is proposed to learn a meaningful and refined representation of GUIs. The VQ-VAE is evaluated in two scenarios: a simple task focused on bounding box reconstruction and GUI element category prediction, and a more complex multimodal setting incorporating additional image, text and color features. For the multimodal framework only, two configurations of the VQ-VAE are proposed, one of which is more complex and could theoretically offer enhanced learning capacity.

For the simple task of bounding box reconstruction and GUI element category prediction, the smaller VQ-VAE is tested on both Rico and FLUID. On Rico, the VQ-VAE offers excellent GUI element category prediction performance and good results for bounding box reconstruction, with an accuracy that is greater than 99% and a mIoU of 0.59, respectively. On FLUID, the VQ-VAE achieves similar

results for GUI element prediction, with an accuracy higher than 99%. However, on FLUID the model exhibits a reduced mIoU of 0.37, but the mIoU results are proven to be influenced by the high presence of small GUI elements in FLUID, for which even the slightest misalignment in position can result in a very low score for the mIoU.

The VQ-VAE is then tested in the multimodal setting only on FLUID, with its two different configurations. The results indicate that there are no substantial differences in performance between the two configurations. Both VQ-VAEs excel at GUI element category prediction and color category prediction, with an accuracy higher than 98%, and are able to deliver good performance for representation of images and text, with a cosine similarity of 0.81 and 0.84, respectively. Unfortunately, the poor bounding box reconstruction capabilities of the VQ-VAE model are confirmed also in the multimodal scenario, with a mIoU of 0.11.

As bounding box reconstruction is the only weak point of the proposed VQ-VAE model for representation learning of GUIs, future works should better address this task to improve the robustness of the learned representation, for example by tuning the hyperparameters involved in the loss computation or by introducing a loss term that is weighted depending on the size of the GUI elements. Finally, the approach will need to be validated in the context of a downstream real-world application.

# Bibliography


[1] Figma Inc. *Figma: The Collaborative Interface Design Tool.* Accessed: 2024-08-09. URL: https://www.figma.com/ (cit. on pp. 1, 7, 8, 15, 17, 18, 22, 30, 43).

[2] Alexandra Hotti, Riccardo Sven Risuleo, Stefan Magureanu, Aref Moradi, and Jens Lagergren. *The Klarna Product Page Dataset: Web Element Nomination with Graph Neural Networks and Large Language Models.* 2024. arXiv: 2111.02168 [cs.LG]. URL: https://arxiv.org/abs/2111.02168 (cit. on pp. 2, 15, 17).

[3] Kotaro Kikuchi, Naoto Inoue, Mayu Otani, Edgar Simo-Serra, and Kota Yamaguchi. *Generative Colorization of Structured Mobile Web Pages.* 2023. arXiv: 2212.11541 [cs.CV]. URL: https://arxiv.org/abs/2212.11541 (cit. on pp. 3, 8, 15–17, 19).

[4] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. «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.* UIST '17. Québec City, QC, Canada: Association for Computing Machinery, 2017, pp. 845–854. ISBN: 9781450349819. DOI: 10.1145/3126594.3126651. URL: https://doi.org/10.1145/3126594.3126651 (cit. on pp. 4, 8, 10, 11, 18, 43, 52).

[5] Jakob Nielsen. *10 Usability Heuristics for User Interface Design.* Accessed: 2024-09-02. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (cit. on p. 5).

[6] Inc. Amazon.com. *Amazon Mobile App.* Accessed: 2024-10-09. URL: https://www.amazon.com (cit. on p. 5).

[7] Interaction Design Foundation. *The Gestalt Principles.* Accessed: 2024-09-02. URL: https://www.interaction-design.org/literature/topics/gestalt-principles (cit. on p. 5).

[8] Atlassian. *Trello App*. Accessed: 2024-10-09. URL: https://trello.com (cit. on p. 6).

[9] Thomas Langerak, Sammy Christen, Mert Albaba, Christoph Gebhardt, and Otmar Hilliges. *MARLUI: Multi-Agent Reinforcement Learning for Adaptive UIs*. 2023. arXiv: 2209.12660 [cs.HC]. URL: https://arxiv.org/abs/2209.12660 (cit. on p. 8).

[10] Toby Jia-Jun Li, Lindsay Popowski, Tom M. Mitchell, and Brad A. Myers. «Screen2Vec: Semantic Embedding of GUI Screens and GUI Components». In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama, Japan: ACM, 2021. DOI: 10.1145/3411764.3445049 (cit. on pp. 8–10).

[11] Forrest Huang, John F. Canny, and Jefrey Nichols. «Swire: Sketch-based User Interface Retrieval». In: *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '19)*. Glasgow, Scotland, UK: Association for Computing Machinery, 2019, pp. 1–10. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300334 (cit. on pp. 8–10).

[12] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. «Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots». In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM, Oct. 2021. DOI: 10.1145/3472749.3474763. URL: http://dx.doi.org/10.1145/3472749.3474763 (cit. on pp. 8–10).

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 8, 38).

[14] Emilio Arroyo, Jun Gao, Mohit Bansal, and Alexander G. Schwing. «Variational Transformer Networks for Layout Generation». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 13632–13641. DOI: 10.1109/CVPR46437.2021.01342. URL: https://openaccess.thecvf.com/content/CVPR2021/papers/Arroyo_Variational_Transformer_Networks_for_Layout_Generation_CVPR_2021_paper.pdf (cit. on pp. 8, 10).

[15] Diederik P Kingma and Max Welling. «Auto-encoding variational Bayes». In: *arXiv preprint arXiv:1312.6114* (2013) (cit. on pp. 8, 11, 36, 37).

[16] Andrey Sobolevsky, Guillaume-Alexandre Bilodeau, Jinghui Cheng, and Jin L. C. Guo. *GUILGET: GUI Layout GEneration with Transformer*. 2023. arXiv: 2304.09012 [cs.CV]. URL: https://arxiv.org/abs/2304.09012 (cit. on pp. 8, 10).

[17] Atsuhiro Yamaguchi, Yijun Li, Tsung-Yi Lin, Zhe Lu, and Wei-Lun Chao. «CanvasVAE: Learning to Generate Vector Graphic Documents». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 8565–8574 (cit. on pp. 9, 10).

[18] Jihoon Lee, Minjung Kim, and Juho Kim. «Neural Design Network: Graphic Layout Generation with Constraints». In: *arXiv preprint arXiv:1912.09421* (2019) (cit. on pp. 9, 10).

[19] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. «The Graph Neural Network Model». In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: `10.1109/TNN.2008.2005605` (cit. on p. 9).

[20] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`. URL: `https://arxiv.org/abs/1406.2661` (cit. on p. 9).

[21] Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. «Constrained Graphic Layout Generation via Latent Optimization». In: *Proceedings of the 29th ACM International Conference on Multimedia*. ACM, Oct. 2021. DOI: `10.1145/3474085.3475497`. URL: `http://dx.doi.org/10.1145/3474085.3475497` (cit. on pp. 9, 10).

[22] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: `1908.10084 [cs.CL]`. URL: `https://arxiv.org/abs/1908.10084` (cit. on pp. 9, 61).

[23] David E. Rumelhart and James L. McClelland. «Learning Internal Representations by Error Propagation». In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1987, pp. 318–362 (cit. on p. 9).

[24] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: `1511.08458 [cs.NE]`. URL: `https://arxiv.org/abs/1511.08458` (cit. on p. 9).

[25] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: `1409.1556 [cs.CV]`. URL: `https://arxiv.org/abs/1409.1556` (cit. on p. 9).

[26] Panupong Pasupat, Tian-Shun Jiang, Evan Zheran Liu, Kelvin Guu, and Percy Liang. «Mapping Natural Language Commands to Web Elements». In: *arXiv preprint arXiv:1808.09132* (2018) (cit. on p. 10).

[27] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. «Mapping Natural Language Instructions to Mobile UI Action Sequences». In: *arXiv preprint arXiv:2005.03776* (2020) (cit. on p. 10).

[28] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. «Learning Design Semantics for Mobile Apps». In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*. Berlin, Germany: ACM, 2018, pp. 569–579. DOI: 10.1145/3242587.3242650 (cit. on p. 10).

[29] Qianzhi Jing, Tingting Zhou, Yixin Tsang, Liuqing Chen, Lingyun Sun, Yankun Zhen, and Yichun Du. «Layout Generation for Various Scenarios in Mobile Shopping Applications». In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581446. URL: https://doi.org/10.1145/3544548.3581446 (cit. on pp. 10, 53).

[30] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. *GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks*. 2021. arXiv: 2101.09978 [cs.HC]. URL: https://arxiv.org/abs/2101.09978 (cit. on p. 10).

[31] Mohammad Amin Mozaffari, Xinyuan Zhang, Jinghui Cheng, and Jin L.C. Guo. «GANSpiration: Balancing Targeted and Serendipitous Inspiration in User Interface Design with Style-Based Generative Adversarial Network». In: *CHI Conference on Human Factors in Computing Systems*. CHI '22. ACM, Apr. 2022. DOI: 10.1145/3491102.3517511. URL: http://dx.doi.org/10.1145/3491102.3517511 (cit. on p. 10).

[32] Naoto Inoue, Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. *LayoutDM: Discrete Diffusion Model for Controllable Layout Generation*. 2023. arXiv: 2303.08137 [cs.CV]. URL: https://arxiv.org/abs/2303.08137 (cit. on p. 10).

[33] Eldon Schoop, Xin Zhou, Gang Li, Zhourong Chen, Björn Hartmann, and Yang Li. *Predicting and Explaining Mobile UI Tappability with Vision Modeling and Saliency Analysis*. 2022. arXiv: 2204.02448 [cs.HC]. URL: https://arxiv.org/abs/2204.02448 (cit. on p. 10).

[34] Tony Beltramelli. *pix2code: Generating Code from a Graphical User Interface Screenshot*. 2017. arXiv: 1705.07962 [cs.LG]. URL: https://arxiv.org/abs/1705.07962 (cit. on p. 10).

[35] Alex Robinson. *Sketch2code: Generating a website from a paper mockup*. 2019. arXiv: 1905.13750 [cs.CV]. URL: https://arxiv.org/abs/1905.13750 (cit. on p. 10).

[36] Erfan Eshratifar. *Variational Auto Encoder (VAE) for the Numerai Dataset*. Accessed: 2024-09-24. URL: https://amirerfan.medium.com/variational-auto-encoders-vae-for-the-numerai-dataset-2709dcc7e449 (cit. on p. 12).

[37] Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. *Attribute2Image: Conditional Image Generation from Visual Attributes*. 2016. arXiv: 1512.00570 [cs.LG]. URL: https://arxiv.org/abs/1512.00570 (cit. on p. 13).

[38] William Harvey, Saeid Naderiparizi, and Frank Wood. *Conditional Image Generation by Conditioning Variational Auto-Encoders*. 2022. arXiv: 2102.12037 [cs.CV]. URL: https://arxiv.org/abs/2102.12037 (cit. on p. 13).

[39] Yang Li, Quan Pan, Suhang Wang, Haiyun Peng, Tao Yang, and Erik Cambria. *Disentangled Variational Auto-Encoder for Semi-supervised Learning*. 2018. arXiv: 1709.05047 [cs.LG]. URL: https://arxiv.org/abs/1709.05047 (cit. on p. 13).

[40] Tom Joy, Sebastian M Schmon, Philip Hilaire Torr, Siddharth Narayanaswamy, and Tom Rainforth. «Rethinking Semi-Supervised Learning in VAEs». In: *arXiv preprint arXiv:2006.10102* (2020) (cit. on p. 13).

[41] Yu Zhou, Xiaomin Liang, Wei Zhang, Linrang Zhang, and Xing Song. «VAE-based Deep SVDD for anomaly detection». In: *Neurocomputing* 453 (2021), pp. 131–140. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.04.089. URL: https://www.sciencedirect.com/science/article/pii/S0925231221006470 (cit. on p. 13).

[42] Syed Sarmad Ali, Jian Ren, and Ji Wu. «Framework to improve software effort estimation accuracy using novel ensemble rule». In: *Journal of King Saud University - Computer and Information Sciences* (2024), p. 102189. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2024.102189. URL: https://www.sciencedirect.com/science/article/pii/S1319157824002787 (cit. on p. 13).

[43] Yunsheng Wang, Xinghan Xu, Lei Hu, Jianwei Liu, Xiaohui Yan, and Weijie Ren. «Continuous imputation of missing values in time series via Wasserstein generative adversarial imputation networks and variational auto-encoders model». In: *Physica A: Statistical Mechanics and its Applications* 647 (2024), p. 129914. ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2024.129914. URL: https://www.sciencedirect.com/science/article/pii/S0378437124004230 (cit. on p. 13).

[44] Md Tahmeed Abdullah, Sejuti Rahman, Shafin Rahman, and Md Fokhrul Islam. «VAE-GAN3D: Leveraging image-based semantics for 3D zero-shot recognition». In: *Image and Vision Computing* 147 (2024), p. 105049. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2024.105049. URL: https://www.sciencedirect.com/science/article/pii/S0262885624001537 (cit. on p. 13).

[45] Ruowei Wang, Yu Liu, Pei Su, Jianwei Zhang, and Qijun Zhao. *3D Semantic Subspace Traverser: Empowering 3D Generative Model with Shape Editing Capability*. 2023. arXiv: 2307.14051 [cs.CV]. URL: https://arxiv.org/abs/2307.14051 (cit. on p. 13).

[46] Xu Tan et al. «NaturalSpeech: End-to-End Text-to-Speech Synthesis With Human-Level Quality». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46.6 (2024), pp. 4234–4245. DOI: 10.1109/TPAMI.2024.3356232 (cit. on p. 13).

[47] Ninon Devis, Nils Demerlé, Sarah Nabi, David Genova, and Philippe Esling. «Continuous Descriptor-Based Control for Deep Audio Synthesis». In: *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2023, pp. 1–5. DOI: 10.1109/ICASSP49357.2023.10096670 (cit. on p. 13).

[48] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018. arXiv: 1711.00937 [cs.LG]. URL: https://arxiv.org/abs/1711.00937 (cit. on pp. 13, 36, 37, 45).

[49] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. *Generating Diverse High-Fidelity Images with VQ-VAE-2*. 2019. arXiv: 1906.00446 [cs.LG]. URL: https://arxiv.org/abs/1906.00446 (cit. on p. 14).

[50] Shiyue Cao, Yueqin Yin, Lianghua Huang, Yu Liu, Xin Zhao, Deli Zhao, and Kaigi Huang. «Efficient-VQGAN: Towards High-Resolution Image Generation with Efficient Vision Transformers». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023, pp. 7368–7377 (cit. on p. 14).

[51] Xueyuan Chen, Xi Wang, Shaofei Zhang, Lei He, Zhiyong Wu, Xixin Wu, and Helen Meng. «Stylespeech: Self-Supervised Style Enhancing with VQ-VAE-Based Pre-Training for Expressive Audiobook Speech Synthesis». In: *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2024, pp. 12316–12320. DOI: 10.1109/ICASSP48485.2024.10446352 (cit. on p. 14).

[52] Sichun Wu, Kazi Injamamul Haque, and Zerrin Yumak. *ProbTalk3D: Non-Deterministic Emotion Controllable Speech-Driven 3D Facial Animation Synthesis Using VQ-VAE*. 2024. arXiv: 2409.07966 [cs.CV]. URL: https://arxiv.org/abs/2409.07966 (cit. on p. 14).

[53] Hao Liu, Wilson Yan, and Pieter Abbeel. «Language Quantized AutoEncoders: Towards Unsupervised Text-Image Alignment». In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 2023, pp. 4382–4395. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/0df1738319f8c6e15b58cb16ea3cfa57-Paper-Conference.pdf (cit. on p. 14).

[54] Zhihao Duan, Ming Lu, Zhan Ma, and Fengqing Zhu. «Lossy Image Compression with Quantized Hierarchical VAEs». In: *2023 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. Vol. 33. IEEE, Jan. 2023, pp. 198–207. DOI: 10.1109/wacv56688.2023.00028. URL: http://dx.doi.org/10.1109/WACV56688.2023.00028 (cit. on p. 14).

[55] Wilson Yan, Yunzhi Zhang, Pieter Abbeel, and Aravind Srinivas. *VideoGPT: Video Generation using VQ-VAE and Transformers*. 2021. arXiv: 2104.10157 [cs.CV]. URL: https://arxiv.org/abs/2104.10157 (cit. on p. 14).

[56] Klarna Incubator. *WebTraversalLibrary*. Accessed: 2024-09-05. 2024. URL: https://github.com/klarna-incubator/webtraversallibrary (cit. on p. 16).

[57] Selenium. *WebDriver — Selenium*. Accessed: 2024-10-09. URL: https://www.selenium.dev/documentation/webdriver/ (cit. on p. 16).

[58] Google. *Google Chrome*. Accessed: 2024-10-09. URL: https://www.google.com/chrome/ (cit. on p. 16).

[59] Builder.io. *Figma to HTML Plugin*. Accessed: 2024-09-05. 2024. URL: https://github.com/BuilderIO/figma-html (cit. on pp. 18, 22, 24).

[60] Cheerio. *Cheerio - The fast, flexible & elegant library for parsing and manipulating HTML and XML*. Accessed: 2024-10-10. URL: https://cheerio.js.org/ (cit. on p. 20).

[61] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *arXiv preprint arXiv:1810.04805* (2019). URL: https://arxiv.org/abs/1810.04805 (cit. on pp. 38, 45, 70).

[62] Łukasz Kaiser, Aurko Roy, Ashish Vaswani, Niki Parmar, Samy Bengio, Jakob Uszkoreit, and Noam Shazeer. *Fast Decoding in Sequence Models using Discrete Latent Variables*. 2018. arXiv: 1803.03382 [cs.LG]. URL: https://arxiv.org/abs/1803.03382 (cit. on pp. 39, 42).

[63] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. 2013. arXiv: 1308.3432 [cs.LG]. URL: https://arxiv.org/abs/1308.3432 (cit. on p. 42).

[64] Python Software Foundation. *Python*. Accessed: 2024-10-09. URL: https://www.python.org/ (cit. on p. 42).

[65] The Linux Foundation. *PyTorch*. Accessed: 2024-10-09. URL: https://pytorch.org/ (cit. on p. 42).

[66] NVIDIA Corporation. *CUDA Toolkit*. Accessed: 2024-10-10. URL: https://developer.nvidia.com/cuda-toolkit (cit. on p. 42).

[67] DigitalOcean. *Paperspace*. Accessed: 2024-10-10. URL: https://www.paperspace.com/ (cit. on p. 42).

[68] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101 (cit. on p. 44).

[69] Alec Radford et al. «Learning Transferable Visual Models From Natural Language Supervision». In: *arXiv preprint arXiv:2103.00020* (2021). URL: https://arxiv.org/abs/2103.00020 (cit. on pp. 46, 61, 66).

[70] Scikit-learn Developers. *sklearn.cluster.KMeans*. Accessed: 2024-10-13. URL: https://scikit-learn.org/1.5/modules/generated/sklearn.cluster.KMeans.html (cit. on p. 48).

[71] Yue Jiang, Changkong Zhou, Vikas Garg, and Antti Oulasvirta. «Graph4GUI: Graph Neural Networks for Representing Graphical User Interfaces». In: *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. ACM. Honolulu, HI, USA, 2024, pp. 1–18. DOI: 10.1145/3613904.3642822 (cit. on p. 51).

[72] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. USA: McGraw-Hill, Inc., 1986. ISBN: 0070544840 (cit. on p. 61).

[73] Daniel Cer et al. «Universal Sentence Encoder». In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2018, pp. 169–174 (cit. on p. 61).