Automating Generation of Web GUI from a Design Image

by

Ajitesh Janardan Singh

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2023 by the
Graduate Supervisory Committee:

Ajay Bansal, Chair
Alexandra Mehlhase
Tyler Baron

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

Frontend development often involves the repetitive and time-consuming task of transforming a Graphical User interface (GUI) design into Frontend Code. The GUI design could either be an image or a design created on tools like Figma, Sketch, etc. This process can be particularly challenging when the website designs are experimental and undergo multiple iterations before the final version gets deployed. In such cases, developers work with the designers to make continuous changes and improve the look and feel of the website. This can lead to a lot of reworks and a poorly managed codebase that requires significant developer resources. To tackle this problem, researchers are exploring ways to automate the process of transforming image designs into functional websites instantly. This thesis explores the use of machine learning, specifically Recurrent Neural networks (RNN) to generate an intermediate code from an image design and then compile it into a React web frontend code. By utilizing this approach, designers can essentially transform an image design into a functional website, granting them creative freedom and the ability to present working prototypes to stockholders in real-time. To overcome the limitations of existing publicly available datasets, the thesis places significant emphasis on generating synthetic datasets. As part of this effort, the research proposes a novel method to double the size of the pix2code [2] dataset by incorporating additional complex HTML elements such as login forms, carousels, and cards. This approach has the potential to enhance the quality and diversity of training data available for machine learning models. Overall, the proposed approach offers a promising solution to the repetitive and time-consuming task of transforming GUI designs into frontend code.

DEDICATION

To my family and friends.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor, Dr. Ajay Bansal, for his guidance and support throughout this research. His insightful feedback and encouragement have been invaluable in shaping the direction and scope of this project. I would also like to thank Dr. Alexandra Mehlhase, Dr, Tyler Baron for their time, expertise and valuable insights during the review and evaluation of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF SYMBOLS AND NOMENCLATURE

3Rs	Reconstruction, Reorganization, and recognition
CSS	Cascaded Style Sheet
CNN	Convolutional Neural Network
CV	Computer Vision
DSL	Domain Specific Language
GAN	Generative Adversarial Network
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
JSON	JavaScript Object Notation
LSTM	Long Short-Term Memory
NLP	Natural Language Processing
OCR	Optical Character Recognition
PBD	Programming by Demonstration
RNN	Recurrent Neural Network
SILK	Sketching Interfaces Like Krazy
UI	User Interface

CHAPTER 1

INTRODUCTION

1.1 Context

Frontend Web development follows an iterative process. The process starts with requirement gathering from the client. Once the requirements are finalized, a designer experiments with various ideas and creates low-fidelity wireframes. These wireframes could be drawn on paper or on design tools like Sketch and Figma. Before the development starts, the client usually goes through the wireframes and approves or recommends changes accordingly. Once the designs are approved, the developers get started with the implementation. Although the design was approved, constant feedback is taken from the client during the development to make sure the final product meets the expectations [16].

There is a slightly optimized procedure followed in established tech companies. Most of the established tech companies develop features or products in-house. This means that a lot of small components are used multiple times across different web pages or products. In order to save engineering resources and reduce duplication of work, they came up with their own design systems. For e.g., Airbnb shared some details of their design system [15] built to make frontend development efficient. A design system with pre-built configurable components reduces the duplicated work by a huge margin. These pre-built components are packages that can be easily plugged and used into a new application. This gives the flexibility of reusing the same component with different configurations in various applications.

1.2 Motivation

Reusing prebuilt components reduces the development time significantly. Although this doesn’t eliminate the developers' involvement in prototyping the ideas proposed by the designers. The major drawback is that most of the ideas get scrapped by the client after the prototyping phase. Engineering time could be better used for making small tweaks if needed and implementing the necessary functionalities. Hence the proposed approach helps in reducing the developer's involvement in the prototyping phase. The ability to transform the image design into a web page instantly gives a lot of power to the designers. Along with that, it gives the designers a lot more flexibility to work with more ideas and get feedback from the client by sharing a real webpage.

Some of the companies like Airbnb [1] have a prebuilt component libraries with their own design language system, allowing engineers to quickly convert mock-ups into functional websites. Hence, the objective of this thesis is to eliminate the manual transformation of mock-ups into code. This goal is easier to accomplish when the mock-ups use a predefined set of components from the component library.

1.3 Problem Statement

There are various approaches to solving this problem. Much research has followed a computer vision approach. The drawback of using a Computer Vision approach is it detects the UI element using contour detection. The contour detection algorithm works in a defined environment and could fail even if the design changes a slightly small feature of a particular element. Extracting elements using contour detection gives a list of elements unordered. The contour detection algorithm doesn’t maintain the layout of the element. To be able to use the auto-generated code as a starter app, the layout needs to be well structured. Otherwise making any edits to the code would be difficult and unpredictable. Even if it detects the UI elements correctly, extracting all the details of the individual component requires another Optical character recognition (OCR) step. This would require a constant update of the algorithm if any new components are added and any changes are made to an existing component type. It is also a very tedious task to implement contour detection to all types of UI elements individually. Each UI element requires some unique feature to be distinguished from the others. Although this approach would give the output accurately for a certain type of GUI interface, it would fail to scale to new elements regularly and easily.

Hence the proposed solution focuses on generating the layout instead of styling the individual components. The styling is handled by using prebuilt components which come with default styles. Once developers get a starter code with correctly defined layouts and prebuilt components, making any changes or customizing becomes very quick.

1.4 Contribution

This research focuses on the generation of intermediate domain-specific language (DSL) from an image design and compiling it into a react code. The Domain Specific Language (DSL) represents the nested structure of elements, where each token maps to a prebuilt React component with the default configuration. The compiler transforms each token into its corresponding React component, maintaining the layout of the components.

During this time of research, only one dataset, open-sourced by pix2code [2] was available. This dataset contained 1750 GUI image designs, each with a corresponding intermediate code. The intermediate code had a sequence of tokens describing the GUI image design. The vocabulary size of these tokens was limited to only 19. Therefore, a methodology was devised to reverse engineer this dataset and expand its size by including more HTML components. This research discusses both the dataset and the methodology used for this expansion.

CHAPTER 2

LITERATURE REVIEW

Although there isn't a lot of research done in the field of automating front-end development, there are a few popular papers that discuss different methodologies for making the front-end development process more efficient.

2.1 Programming by Demonstration (PBD)

Programming by demonstration (PBD) is a technique where the designers can create an end-to-end flow of mock-up designs. One of the earliest research was done by James A Landay and Brad A Myers [11] in which they proposed the tool SILK, which stands for Sketching Interfaces Like Krazy. This research focused on making the design process more efficient. SILK could detect gestures and recognize widgets (Figure 1). Also, it provided various gestures for common actions like copying and grouping. For recognizing the widget from gestures/drawings, the engine used Rubine's gesture recognition algorithm [12].

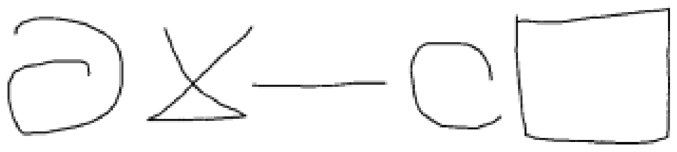


Figure 1: Gesture for cycling, deleting moving, copying and grouping. Data Source: Landay, J.A. (1995). Interactive sketching for user interface design. In CHI '95 Conference Companion on Human Factors in Computing Systems (pp. 47).

SILK was one of the earliest examples of Programming-by-demonstration (PBD). This was introduced in SILK as a behaviour mode where the designers could play around with the layout and structure of the interface. The final design included flows where you could navigate between screens by clicking buttons/links. This is very similar to how a user would navigate through a website. Programming-by-demonstration gave non-programmers a good tool to create mock-up designs that demonstrated the end-to-end behaviour of a real website. While the mock-up requires the installation of a separate application for the client to access, it may not provide the full experience of a fully functioning website preview.

2.2 Heuristic Methodologies

Heuristic methodologies uses simplified strategies and approximations to extract HTML components using contour detection.

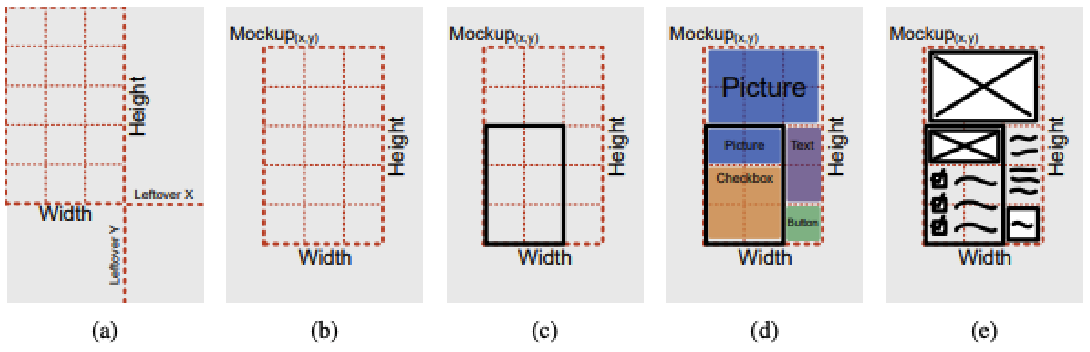
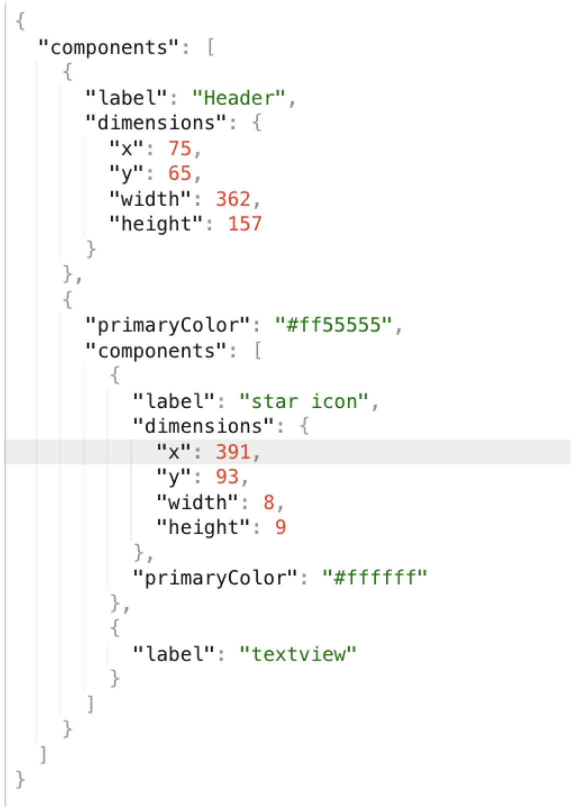


Figure 2: High-level process overview, executed in a sequential order. [Live Web prototype from hand-drawn mock-ups]

One of the research done was extracting components from a Graphical User Interface design and classifying them into respective elements using heuristic methods

[13]. This approach uses Image processing techniques to extract various component positions for a mobile-based application (Figure 2) and create a nested JSON structure. It first detects all the texts in the image using canny edge detection and masks the text. Once the texts are masked, it detects common components like Buttons, Inputs, Text Views, etc using Contour detection techniques. After detecting the common components, it is classified into specific UI component types using a Machine Learning classification model. The next step following the determination of the User Interface (UI) element type is extracting details like text, font type, font size, etc. This is achieved using the Optical Character Recognition (OCR) algorithm. It is finally processed and compiled into a nested JSON structure (Figure 3) which holds all the necessary meta-information relevant to each component along and the component structure.



```
{
  "components": [
    {
      "label": "Header",
      "dimensions": {
        "x": 75,
        "y": 65,
        "width": 362,
        "height": 157
      }
    },
    {
      "primaryColor": "#ff55555",
      "components": [
        {
          "label": "star icon",
          "dimensions": {
            "x": 391,
            "y": 93,
            "width": 8,
            "height": 9
          },
          "primaryColor": "#ffffff"
        },
        {
          "label": "textview"
        }
      ]
    }
  ]
}
```

Figure 3. Nested component structure of website GUI

In the final step, the JSON is parsed and compiled into HTML and CSS. This procedure highly depends on contour detection and works well only for predefined types of UI elements. A slight change in the Sketch Image design might incorrectly detect different contours. For e.g. buttons have various border radii, and detecting contours with all the variations using Image processing would be a challenging task. Hence this process of Image processing and classification, followed by a compiler is not a very flexible approach.

Style-aware sketch-to-code [7] provides a methodology similar to the Extraction and segmentation of graphical user interfaces (Figure 4).

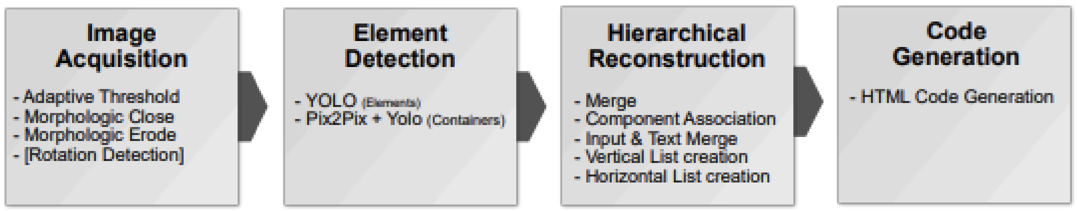


Figure 4: Pipeline processing steps executed sequentially. [Extraction and Classification of User Interface Components from an Image. International Journal of Pure and Applied Mathematics]

It solves the problem in a 3-step process. First, extract components from the image, Second, extract styles from each component. Finally, convert the component into an HTML and CSS code. To segment each component into categories, computer vision algorithms are used. It detects various types of Graphical User Interface elements like buttons, navbars, etc. Once the type of the component is determined, colour extraction and text extraction are done to get more details about the component. Finally, the extracted component features are given as input to a Multi-headed VGG Convolutional

Network. The positional details and the output of the network are used to generate the final code.

Heuristic methodology used for extracting HTML elements from design image primary relies on computer vision contour detection algorithms which can be unreliable in certain situations. For instance, with more complex HTML elements or gradients, contour detection can result in mismatch between different components. For example, a text button could be misidentified as a plain text because geometrically both have text without a bounding box. As a result, it is important to consider the limitations of this approach and potentially supplement it with other techniques to improve reliability.

2.3 Machine Learning methodologies

The state of the art changed completely when Airbnb introduced an internal tool for generating code from low-fidelity wireframes. So far the research done in this field was academic with little to no practical adaptation of it. Although Airbnb didn't open source any implementation details or the dataset, they released an article with a prototype of the system which could create React code from hand-drawn wireframes.

Most of the top tech companies have their own User Interface (UI) design systems (Figure 5). These design systems are followed to keep the consistency in the theme and the design across various apps on different platforms. Also, many times the designers are experimenting with different layouts for a new idea, and prototyping each layout from a developer is a costly and time-consuming process. Also, designing the same wireframe on a design tool like a sketch is a cumbersome and repetitive task. They automated the process of hand-drawn wireframe to a prototype in react using an open-source machine learning algorithm. The total number of common components Airbnb had in its design system at that time was 150. The project is still an ongoing exploration and has a good potential of creating a commercial tool that could assist all the front-end developers to prototype faster.

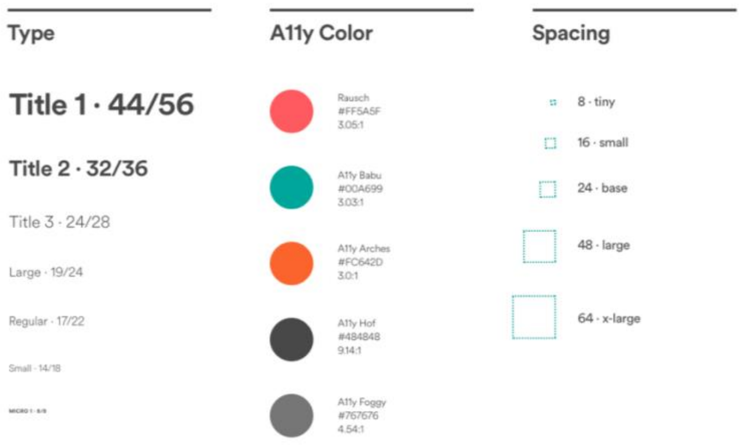


Figure 5: Airbnb Design System.

One of the major problems in this field of study is the scarcity of open source datasets. Even Airbnb hasn't open-sourced the dataset. One of the major contributions done by Tony Betranelli in his research of pix2code [2] was creating and open-sourcing a synthetic dataset. The dataset has about 1750 images of Graphical user interface images and corresponding intermediate code which represents the component structure in the image. pix2code uses an Image captioning approach where GUI screenshots are the inputs and the intermediate code (Figure 6) is predicted similar to caption generation.

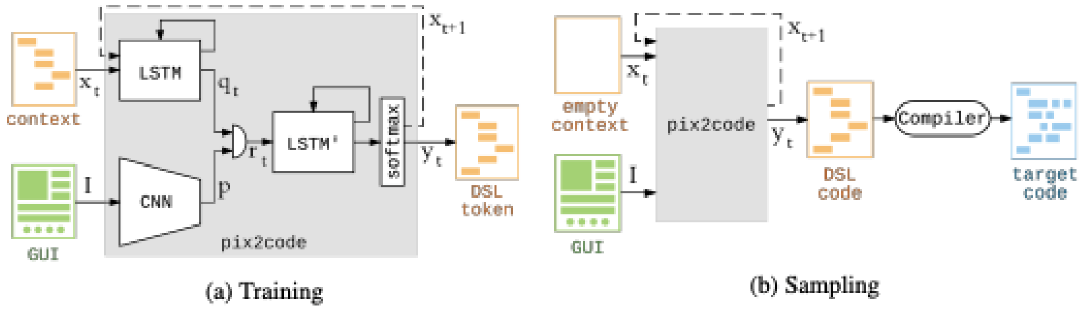


Figure 6: Overview of the pix2code model architecture. [arXiv:1705.07962 [cs.LG]]

This methodology first extracts all the features from an Image using layers of Convolutional Neural networks and then a Long Short term model to predict tokens that represent the intermediate code. 3 different datasets were contributed by Tony Betranelli from his research, each targeting different platforms i.e. web, IOS, and android. All the datasets have a similar intermediate language format. The native code is generated by a compiler for the respective platforms. Instead of using the actual HTML tags, the dataset has 1750 images which coverts 19 types of HTML elements. This was done so that the model could be trained easily because HTML elements usually have an opening and a closing tag which is kind of redundant for an ML model and increases complexity. The CNN layers and the one hot encoded tokens of the language model are given as inputs to the encoder. The LSTM (Figure 6) model is the decoder that predicts the next token for the corresponding input image and the input sequence of tokens.

One other research work built on top of pix2code [2] was Transforming hand-drawn (Figure 7) wireframes into Front end code [7]. The motivation for this work was to simplify the design process for graphic designers, enabling them to create frontend code from hand-drawn sketches. Building on the success of generating HTML and CSS code from screenshots, the authors sought to extend the capabilities of the model to include hand-drawn sketches. To accomplish this, a modified version of pix2code (Figure 8) dataset was utilized.

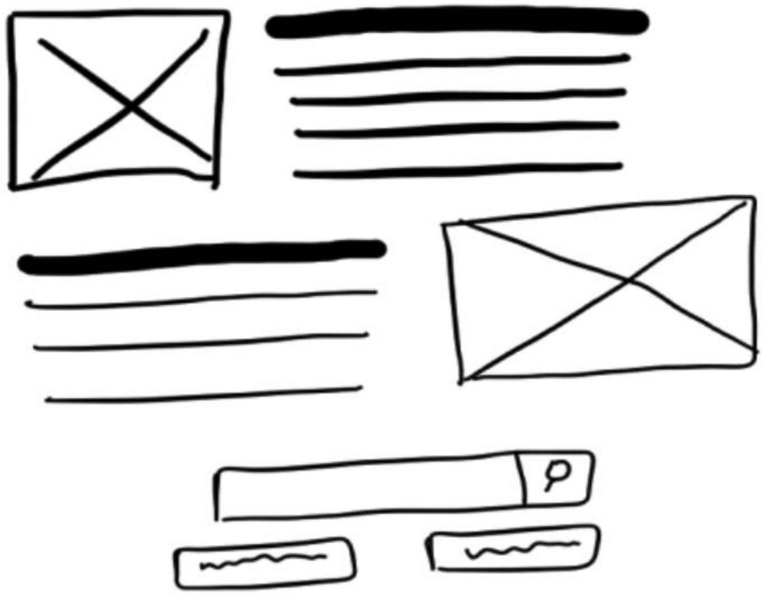


Figure 7: Representation of a hand-drawn user interface design for a website [Transforming hand drawn wireframes into front-end code with deep learning," Computers, Materials & Continua, vol. 72, no.3,pp. 4306]

As you can see the layout could be easily inferred from the image. Hence this becomes an input to the CNN which extracts all the features along with a language model. The rest of the process is pretty similar to the encoder-decoder model used by pix2code.

Another research done by Noah Gundotra was code2pix [6] which was the reverse of what pix2code had achieved. Code2Pix [6] proposed a methodology to generate a graphical user interface from an intermediate language in text format. The Idea of code2pix was suggested by Tony (creator of pix2code) to complete the GAN architecture (Figure 7) where pix2code is the generator and code2pix is the discriminator. code2pix essentially works like a web browser. Similar to a browser, code2pix parses the code to generate a visual representation. The difference to note is browser renders the elements based on predefined rules whereas code2pix achieves a similar result with a deep learning model.

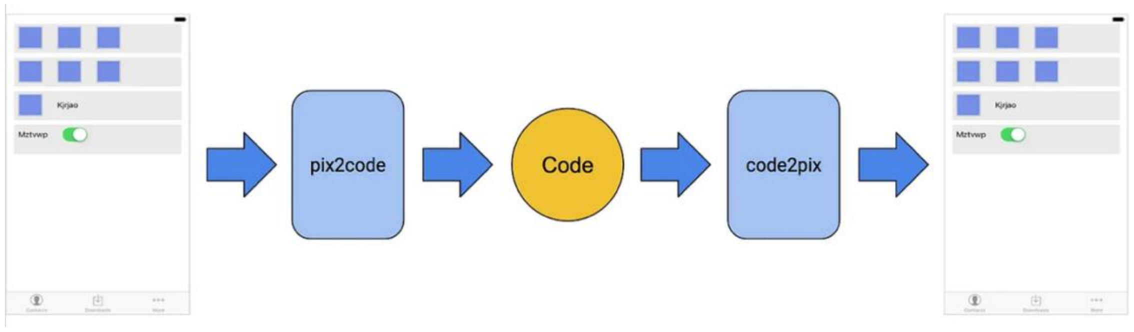


Figure 8: Pix2Code GAN architecture. [code2pix]

The benefit of having a deep learning model for this is it could propagate error signals to improve the renderer without the addition of a new set of rules which is a cumbersome task in a normal renderer. This methodology trains an autoencoder which also serves as an image decoder. The standard autoencoders were not producing the desired result in this case. Hence code2pix created a new architecture called the multi-headed model which world well with all the different types of datasets created by Tony in the pix2code project.

Pix2code is a powerful tool that can convert an image design into an HTML page with a small dataset. However, it has some limitations, as it does not support popular frameworks like React or Angular, and its implementation is not easily extendable. To overcome these challenges and create a machine learning system that can generate functional website GUI code in React. Two methodologies are discussed in this thesis:

first, the technique to generate synthetic dataset (Chapter 3) and second, the methodology used to transform images into React application (Chapter 5).

CHAPTER 3

DATASET

This chapter provides an overview of the existing dataset and proposes an approach for its expansion.

3.1 Pix2code dataset

The Pix2code [2] dataset open-sourced by Tony Betranelli was used as a starting point for our research. Pix2code has a dataset for IOS, Android, and the web. For each platform, there are 1750 images of GUI interfaces and corresponding Domain Specific language (DSL). Since the scope of our research was limited to the web, the dataset size was 1750. The input is an GUI image design (Figure 9) with various HTML elements with an intermediate code in the form of domain specific language (DSL).

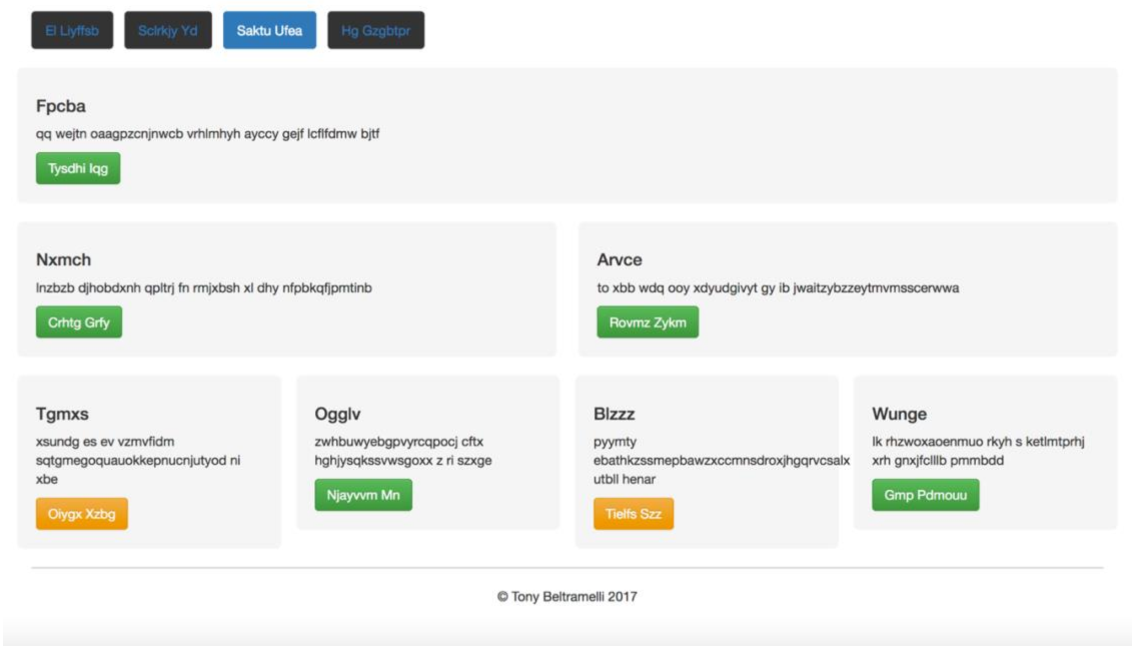
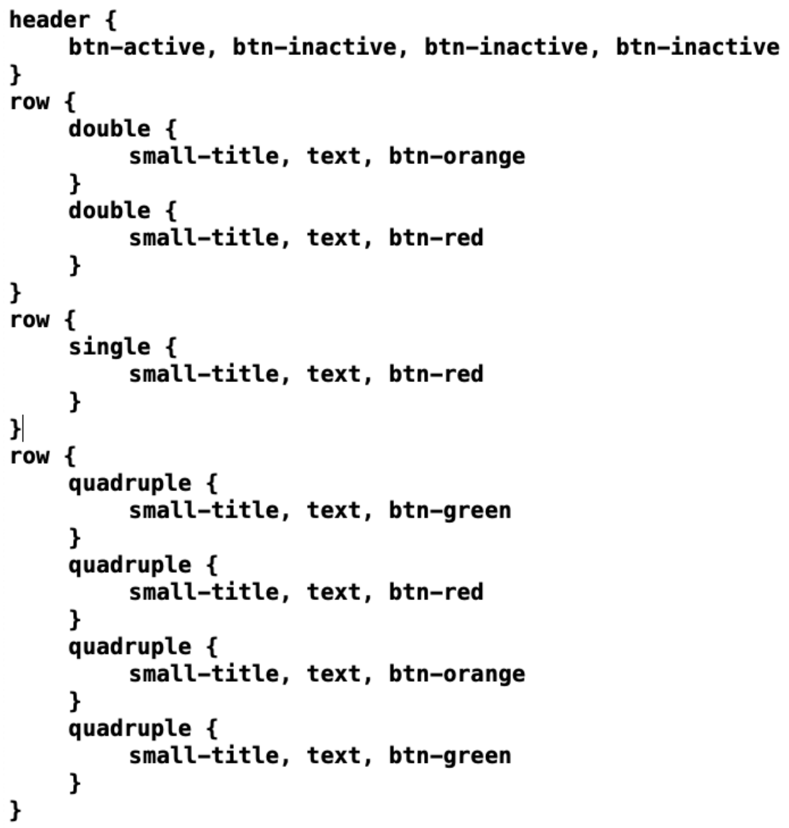

El Liyffsb
Sclrkjy Yd
Saktu Ufea
Hg Gzgbtpr
Fpcba
qq wejtn oaagpzcnjnwcb vrhlmhyh ayccy gejf lcflfdmw bjtf
Tysdhi lqg
Nxmch
lnzbzb djhobdxnh qpltrj fn rmjxbsh xl dhy nfpbkqfjpmtinb
Crhtg Grfy
Arvce
to xbb wdq ooy xdyudgivyt gy ib jwaitzybzzeytmvmsscerwwa
Rovmz Zykm
Tgmxs
xsundg es ev vzmvfidm sqtgmegoquauokkepnucnjutyod ni xbe
Oiygx Xzbg
Ogglv
zwhbuwyebgpvyrcqpocj cftx hghjysqkssvwsgoxx z ri szxge
Njayvvm Mn
Blzzz
pyymty ebathkzssmepbawzxccmnsdroxjhgqrvcsalx utbll henar
Tielfs Szz
Wunge
lk rhzwoxaoenmuo rkyh s ketlmtprhj xrh gnxjfclllb pmmbdd
Gmp Pdmouu
© Tony Beltramelli 2017

Figure 9: GUI Image design



```
header {
    btn-active, btn-inactive, btn-inactive, btn-inactive
}
row {
    double {
        small-title, text, btn-orange
    }
    double {
        small-title, text, btn-red
    }
}
row {
    single {
        small-title, text, btn-red
    }
}
row {
    quadruple {
        small-title, text, btn-green
    }
    quadruple {
        small-title, text, btn-red
    }
    quadruple {
        small-title, text, btn-orange
    }
    quadruple {
        small-title, text, btn-green
    }
}
```

Figure 10: Representation of Domain Specific language

The important detail in the domain specific language is that the it has a nested structure (Figure 10) that holds the layout of the image design. For e.g., The header has 4 buttons with one active and others inactive. Similarly, the second row holds two containers with 3 leaf components. This output could be parsed and compiled into a final front-end code.

The vocab size of the dataset is 19. An RNN/LSTM model takes partial sequence as part of the input and predicts the next token. Hence if each image is sampled into 60 sequences of tokens. The total size of the dataset is 1750*60 = 1,05,000. Apart from the 19 tokens of the vocab, the training is prepended and appended with start and end tokens. The start and the end token are used to make first prediction and terminate the program respectively. A prediction of an end token by the Long Short-term model (LSTM) signifies that all the necessary information has been retrieved from the Inputs.

3.2 Motivation to expand pix2code dataset.

Pix2code dataset size was small, and it had only simple HTML elements. Hence, it managed to achieve an accuracy of 99%. At the time of this research, no other dataset was available. Rather than generating a completely new dataset from scratch, it was decided to expand the existing Pix2code dataset by incorporating more complex HTML elements. This approach was preferred due to the already established good structure of Pix2code, which ultimately saves time and effort.

3.3 Reverse engineering pix2code dataset

To expand the dataset, the initial step was creating a mechanism to regenerate the Pix2code dataset. A methodology (Figure 11) was devised to capture all the necessary information for inputs and outputs of a single GUI design into a JSON format. Although contour detection is considered unreliable, it was utilized to reverse engineer the Pix2code dataset since it only contains low-complexity elements. The sole purpose of contour detection in this context was to extract all the elements from the image design, which is a limited case for the Pix2code dataset.

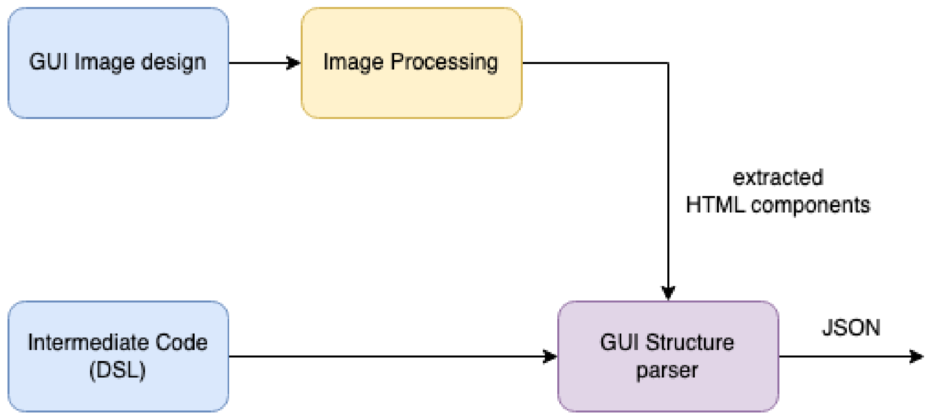


Figure 11: Input Image and Intermediate Code to Structured JSON

The data conversion from image and intermediate code to JSON is a two-step process

1. Extract all the GUI elements from the input image.
2. Parse through the DSL and store the details in a nested JSON format.

Step 1: Extracting GUI elements

Computer Vision is used to clip all the components as a separate image (Figure 12) along with the positional details. Contour detection [23] is the most preferred way to detect objects and localize them. To be able to find contours accurately, it is important that the image has only a single channel. Hence, the first step in contour detection is to convert the image from BGR to grayscale. Further processing of the grayscale image is required to detect the borders of the object of interest. There are two popular mechanisms to highlight the borders of the grayscale image with white pixels.

- Canny edge detection
- Binary thresholding

Pix2code dataset has a white background which helps because thresholding step could be skipped the aforementioned thresholding steps and just invert the grayscale image and make the object of interest lighter. The find contours algorithm ignores the black pixel and detects contours by detecting similar-intensity pixels.

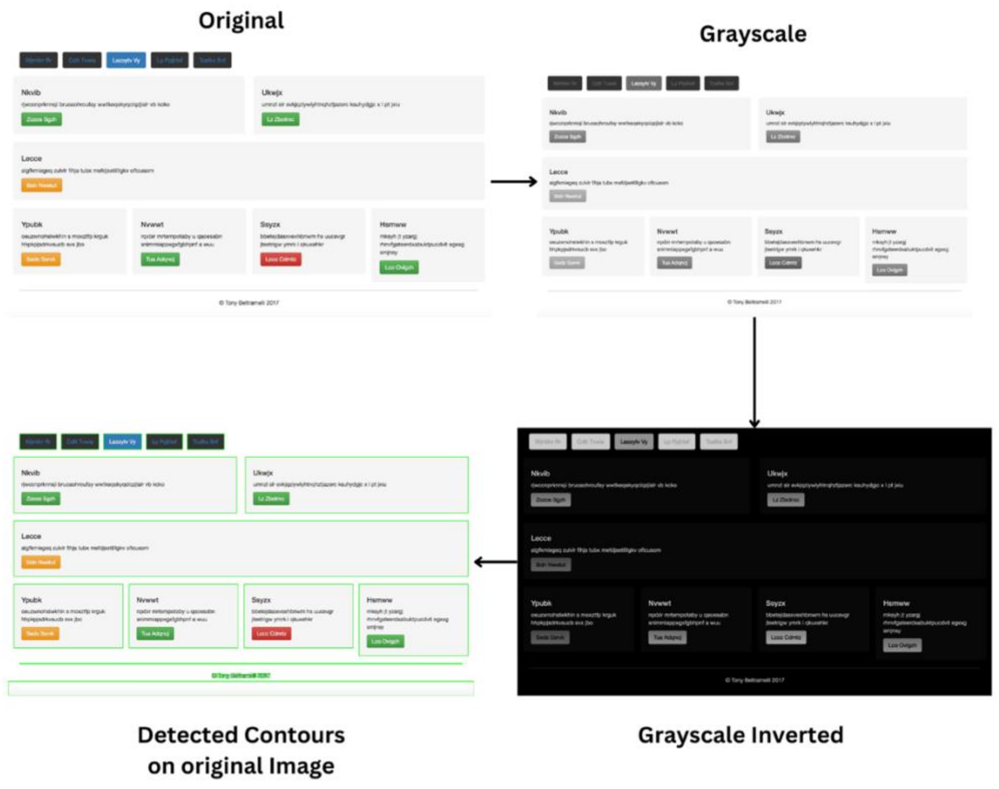


Figure 12: Extracting GUI Elements from Input Image

The feature extraction steps give a list of all the Components with their respective position (Figure 7). The extracted components has a list of all the HTML components with the positions.

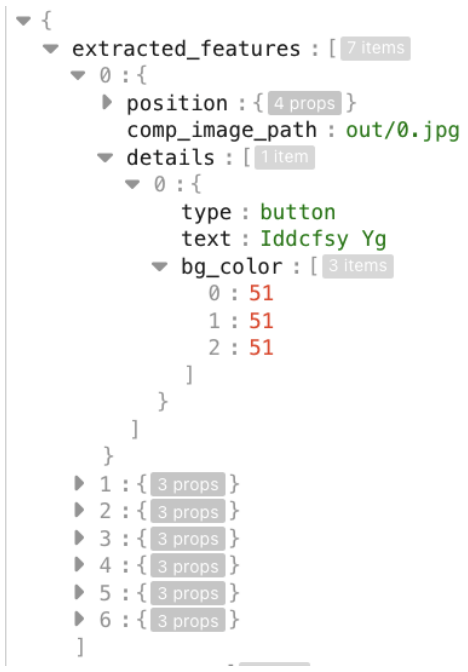

```
{
  extracted_features : [ 7 items
    0 : {
      position : { 4 props }
      comp_image_path : out/0.jpg
      details : [ 1 item
        0 : {
          type : button
          text : Iddcfsy Yg
          bg_color : [ 3 items
            0 : 51
            1 : 51
            2 : 51
          ]
        }
      ]
    }
    1 : { 3 props }
    2 : { 3 props }
    3 : { 3 props }
    4 : { 3 props }
    5 : { 3 props }
    6 : { 3 props }
  ]
```

Figure 13: Features Extracted from a GUI Image design.

Step 2: Parse through the DSL and store the details in a nested JSON format.

This step generates the JSON (Figure 13) from Domain Specific Language (DSL) and extracted components from previous steps. The DSL structure that holds the layout structure and the GUI component type is parsed and mapped with the extracted components list from the previous step. The extracted features from the previous steps provides details of visual representation and the position of the respective GUI element.

This process finally creates a nested JSON which holds all the information of the input image and the output DSL. The idea of reverse engineering the dataset was to be able to generate input and output again from the JSON.

With the help of the aforementioned methodology, we were able to generate 1750 JSON files. The next task was to create a pix2code dataset again from these JSON files and compare if the generated dataset matches the original dataset. The original dataset was regenerated accurately which proved the feasibility of generating synthetic data from structured nested JSONs. Figure 14 shows a high level architecture of Data generator.

.

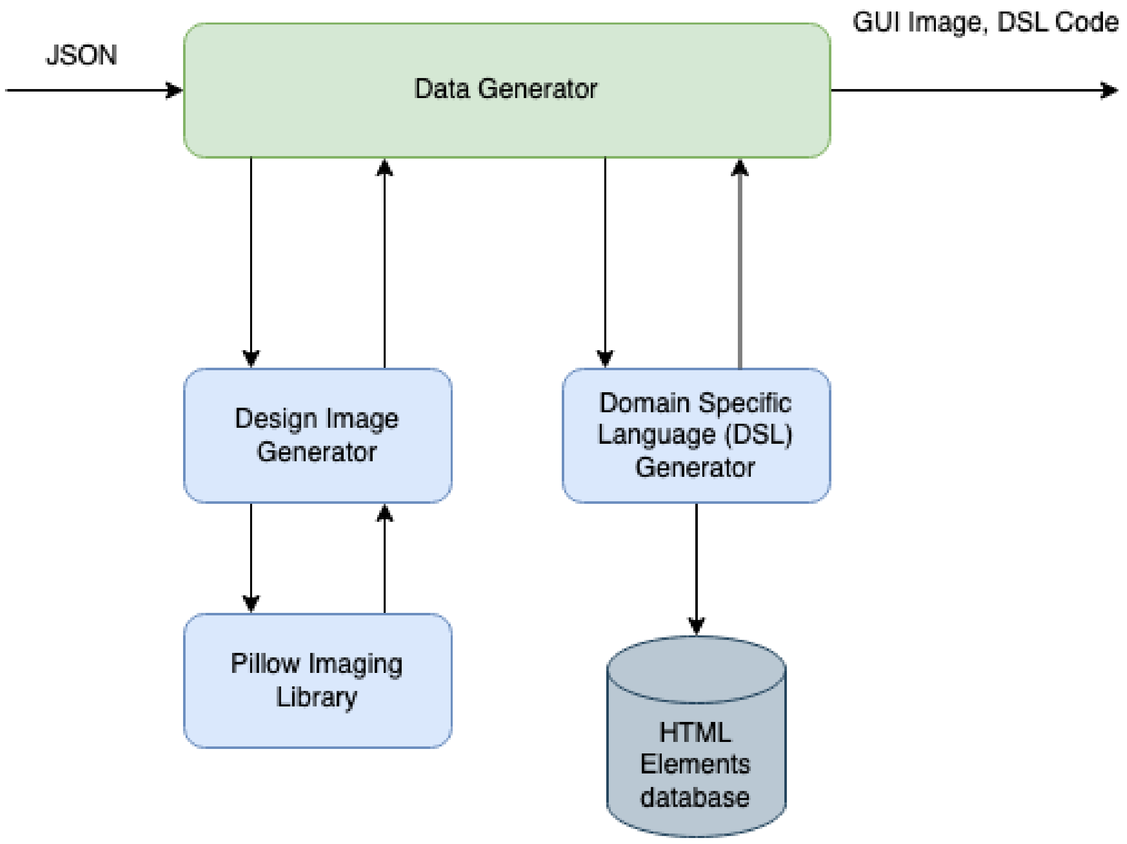
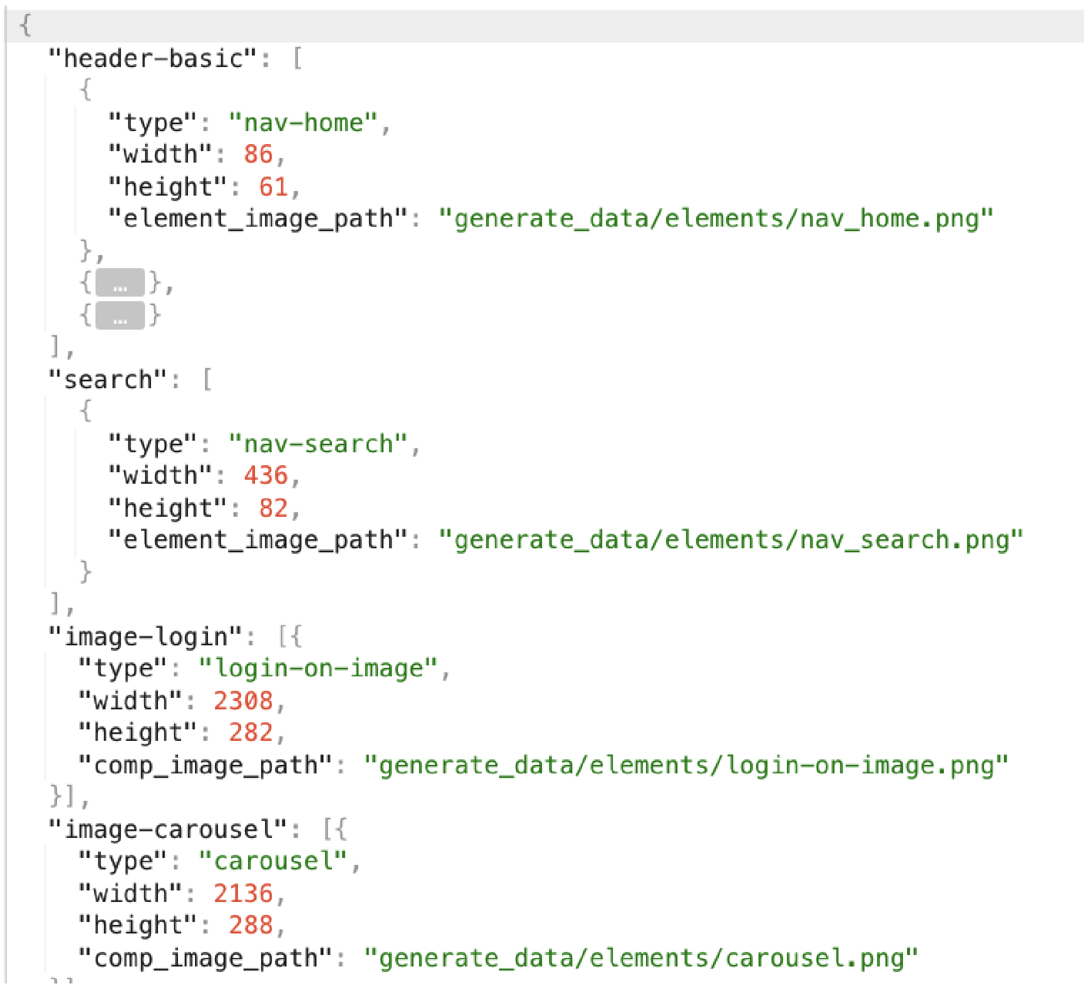


Figure 14: Data regeneration from JSON.

3.4 Methodology to expand dataset.

From the previous step, the mechanism to create synthetic GUI image and intermediate Domain Specific Code (DSL) from JSON is in place. This gives the flexibility to add more elements by just programmatically editing or creating new components in the

JSON constructed in the step. As a proof of concept, several more different types of elements were added to the dataset.



```
{
  "header-basic": [
    {
      "type": "nav-home",
      "width": 86,
      "height": 61,
      "element_image_path": "generate_data/elements/nav_home.png"
    },
    { … },
    { … }
  ],
  "search": [
    {
      "type": "nav-search",
      "width": 436,
      "height": 82,
      "element_image_path": "generate_data/elements/nav_search.png"
    }
  ],
  "image-login": [{
    "type": "login-on-image",
    "width": 2308,
    "height": 282,
    "comp_image_path": "generate_data/elements/login-on-image.png"
  }],
  "image-carousel": [{
    "type": "carousel",
    "width": 2136,
    "height": 288,
    "comp_image_path": "generate_data/elements/carousel.png"
```

Figure 15: HTML Elements database

To create new datasets, an algorithm parses through all the JSON created from the pix2code reverse engineering step and edits/updates/adds new elements to create a new updated JSON. To add/edit new elements to the JSON, random HTML elements are picked from the database (Figure 15). The data generator creates all possible iterations to create a versatile dataset. Adding more elements to the database could help in generating more data with complex elements. Using this methodology, the dataset size was doubled.

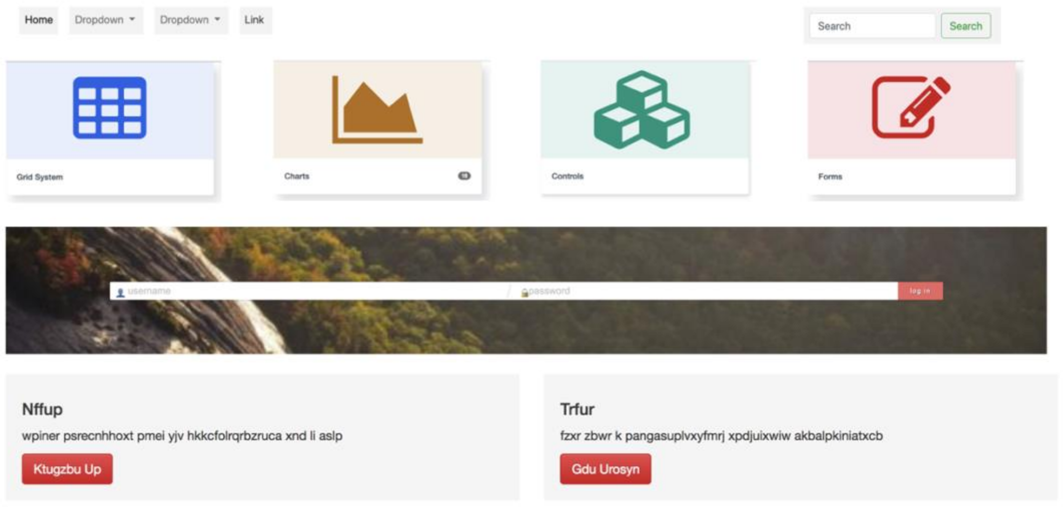
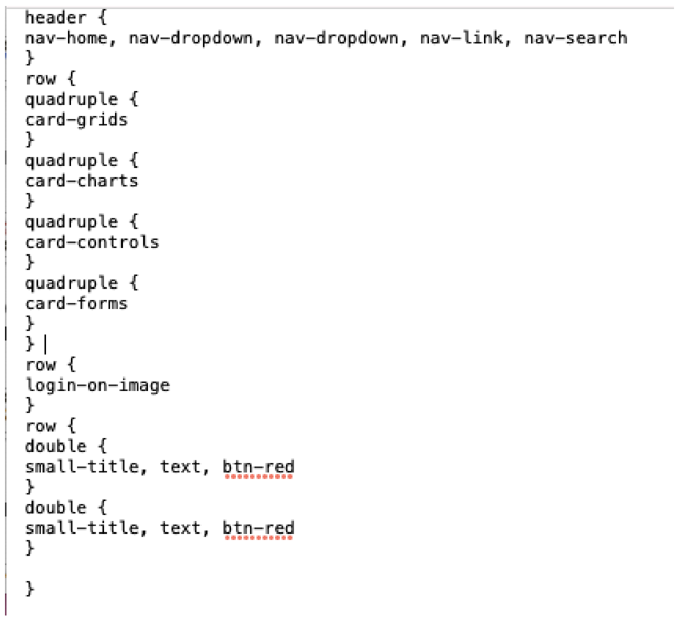


Figure 16 New dataset input Image



```
header {
nav-home, nav-dropdown, nav-dropdown, nav-link, nav-search
}
row {
quadruple {
card-grids
}
quadruple {
card-charts
}
quadruple {
card-controls
}
quadruple {
card-forms
}
}
row {
login-on-image
}
row {
double {
small-title, text, btn-red
}
double {
small-title, text, btn-red
}

}
```

Figure 17: New dataset Domain Specific language.

The structure of the Domain specific language is same as pix2code[2] dataset which is helpful because we could use the original pix2code dataset as well. Figure 16 and 17 show example of an input image and corresponding Domain Specific Language (DSL) generated with new HTML elements. The size of the synthetic dataset generated was 3500 websites.

CHAPTER 4

BACKGROUND

Before delving into the methodology of transforming design image to React code, this section provides an essential overview of relevant concepts.

4.1 Natural Language Processing

Natural language processing [24] is a branch in Artificial Intelligence that focuses on understanding and interpreting human languages in the form of text and speech. Traditionally computers were designed to operate on predefined rules. Computers will throw errors if the program deviate from these rules. Hence, training a computer to understand human language is a difficult task because human language is very ambiguous. A Similar sentence can have different meanings depending on the context it is used.

Natural language processing is implemented using a combination of various techniques. Instead of using a collection of rules in traditional approach, Natural Language processing uses textual data as an input to the machine learning algorithm that creates rules by itself. It combines traditional rule-based modeling with machine learning and deep learning models. The choice of technique or algorithm depends on the problem being solved.

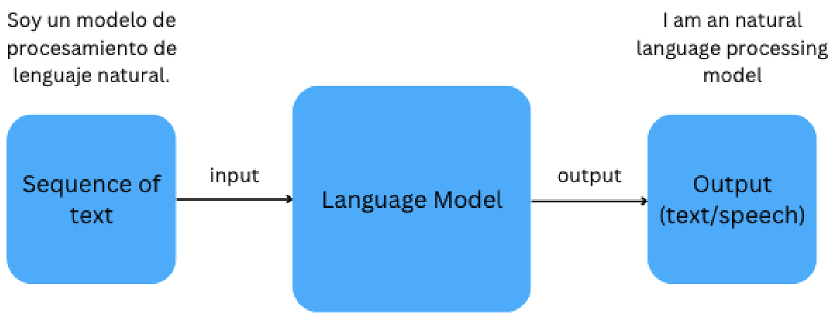

Soy un modelo de procesamiento de lenguaje natural.
Sequence of text
input
Language Model
output
I am an natural language processing model
Output (text/speech)

Figure 18: High level framework of an NLP application.

Figure 18 illustrates that the input feature to a language mode is only a sequence of text which generates an output that may consist of either text or speech audio.

4.2 Computer Vision

Computer Vision is a branch of Artificial Intelligence which works with visual inputs. The visual input could be an Image or a Video. The goal of computer vision is to train a system to be able to understand and interpret image information just like how humans do. Most of the human perception relies on visual input from the eyes. Therefore, to enable computers to perform tasks like humans, it is essential to have the capability to extract important information from images. This be achieved using various approaches like image processing and machine learning.

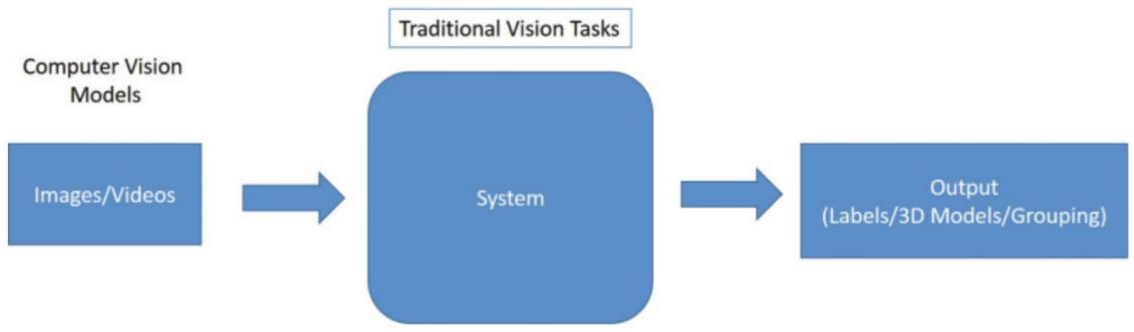


Figure 19: Computer Vision framework which passively outputs language. Data Source Wiriyathammabhum, P., Summers-Stay, D., Fermüller, C., & Aloimonos, Y. (2016). Computer Vision and Natural Language Processing. ACM Computing Surveys (CSUR), 49, 1 - 44.

Traditionally many applications used only the visual feature as an input to predict the attributes into classes (Figure 19). Image processing [25] involves a series of steps, with each step typically involving the use of different algorithms. The first step is pre-processing the image. Pre-processing removes all the unnecessary information from the image, reduces noise and enhances the vital part of the image. Techniques such as Image smoothing, Image sharpening, Image normalization, Image resizing are used for pre-processing. The next step is extracting information from the image, which may involve techniques such as object detection, object recognition, image segmentation, edge, or blob detection etc. Finally, the extracted information is used to interpret the meaning of the image.

Image processing is also used in enhancing the image apart from extracting features. Some of the application of image enhancements are improving image resolution, fixing contrast or brightness, image restoration, object detection etc.

4.3 Natural Language Processing with Computer Vision

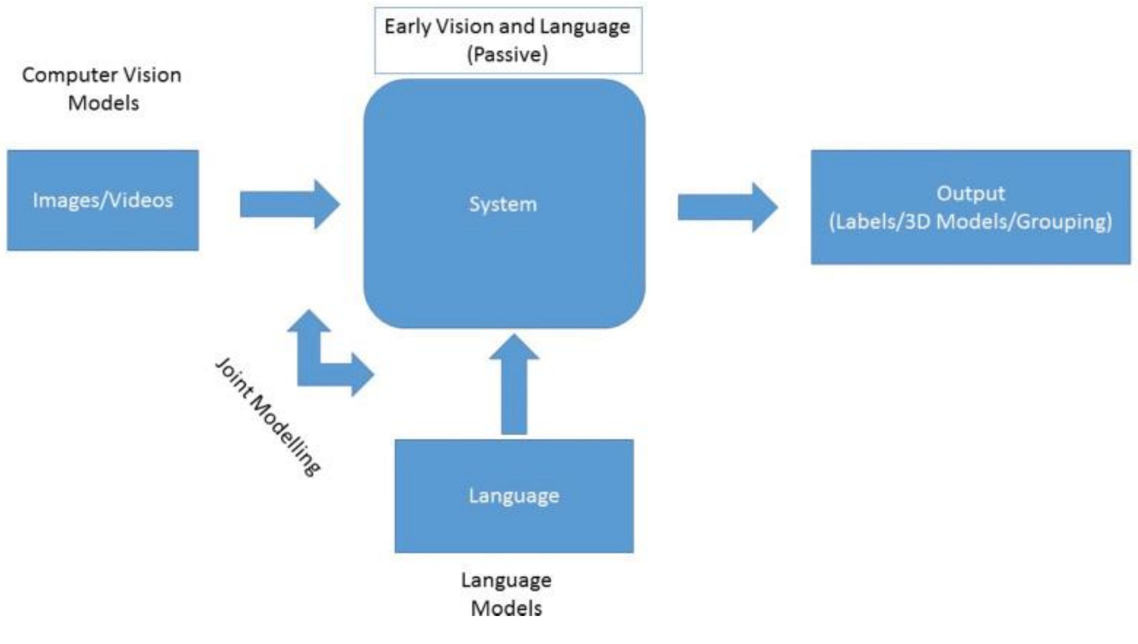


Figure 20 Vision framework which uses language information as an addition input feature. Data Souce: Wiriyathammabhum, P., Summers-Stay, D., Fermüller, C., & Aloimonos, Y. (2016). Computer Vision and Natural Language Processing. ACM Computing Surveys (CSUR), 49, 1

Natural Language processing are used in applications which involve processing text input, while Computer Vision is used in applications that deal with image input. However, there are certain applications like Image captioning that require both image and text as input. By combining features from both image and text data (Figure 20), more accurate predictions can be made. This enables to solve a wide range of problems. Some of the example of NLP with Computer Vision applications are Image and Video Captioning, Visual retrieval, visual question answering, human robot interaction, robotic actions and robot navigation [16].

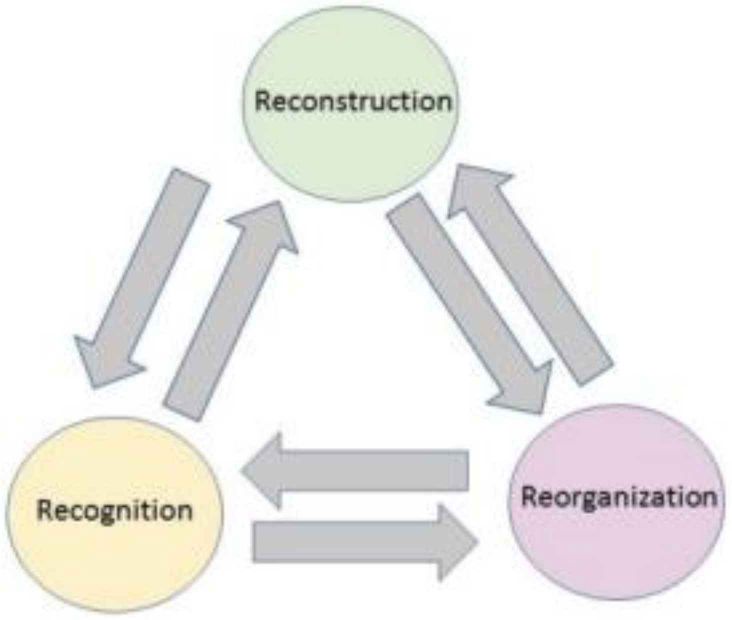


Figure 21: The 3Rs in computer vision [Computer Vision and Natural Language Processing. ACM Computing Surveys (CSUR)]

The 3Rs (Figure 21) proposed by Malik et al [17] which refers to Reconstruction, reorganization and recognition are considered as the central problem in computer vision.

Reconstruction involves generating the 3D model scene from one or more images. This can include task such as Scene reconstruction and Structure from motion. Recognition refers to the process of identifying and labelling objects in an image. This can involve labelling both 2D and 3D objects. Examples of 2D object recognition are handwriting or face recognition. Reorganization means segmenting raw pixels into different classes which helps in extracting meaningful objects from the image. This is achieved using low level and high level image processing. Low level techniques are edge detection, contour detection and corner detection whereas High level techniques involve semantic segmentation into regions and assigning labels to each one. For e.g. Detecting cars, traffic signs, buildings, roads etc.

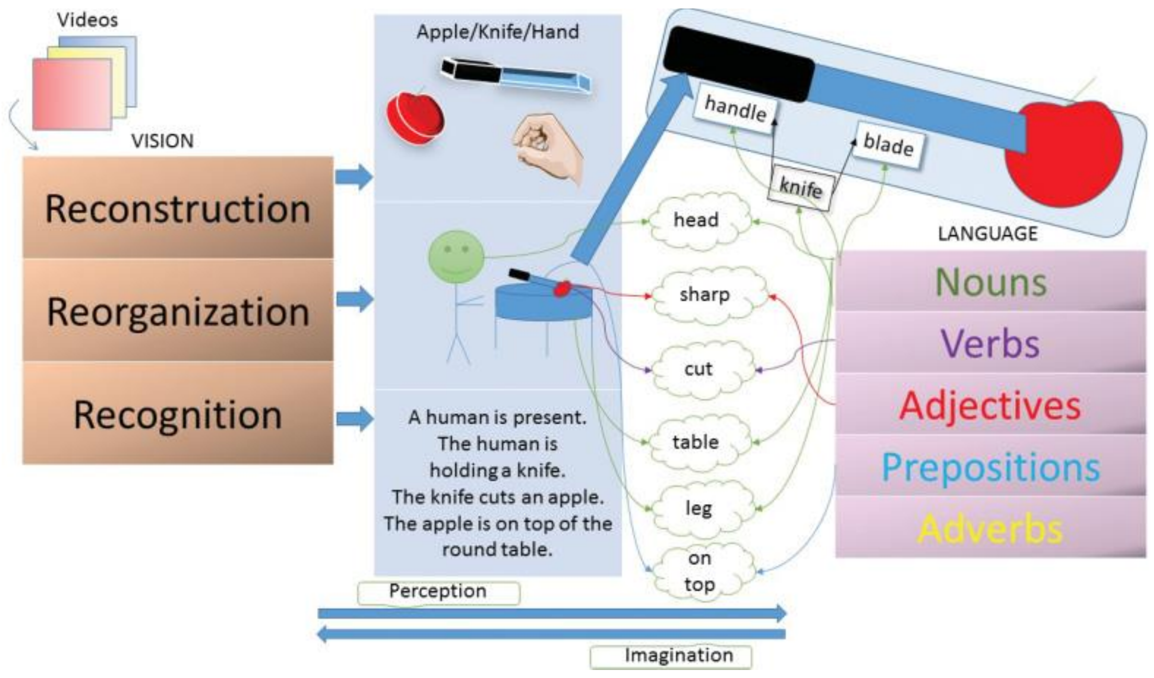


Figure 22: Connecting the 3Rs Data Source: Wiriyathammabhum, P., Summers-Stay, D., Fermüller, C., & Aloimonos, Y. (2016). Computer Vision and Natural Language Processing. ACM Computing Surveys (CSUR), 49, 1 - 44.

The scene in Fig 22 shows a man cutting an apple. In the reorganization phase, the raw pixels are labelled into head, sharp, cut, table, leg, on top etc. These reorganized objects are reconstructed into a scene with an apple, hand and a knife. The reconstructed scene is recognized into a sequence of text. A lot of use cases have varying length of inputs and outputs. These sort of use cases are solved using an encoder-decoder architecture.

4.4 Encoder-Decoder architecture

The encoder-Decoder [29] model is a very popular mechanism to predict a varying length of text. For e.g. Language Translation, Image captioning, etc. In this architecture, the encoder extracts features or necessary information from inputs and converts them into a fixed length vector. The output vector of an encoder is passed as an input to the decoder to predict the sequence of text describing the inputs. This is the most widely used architecture for most of the NLP tasks.

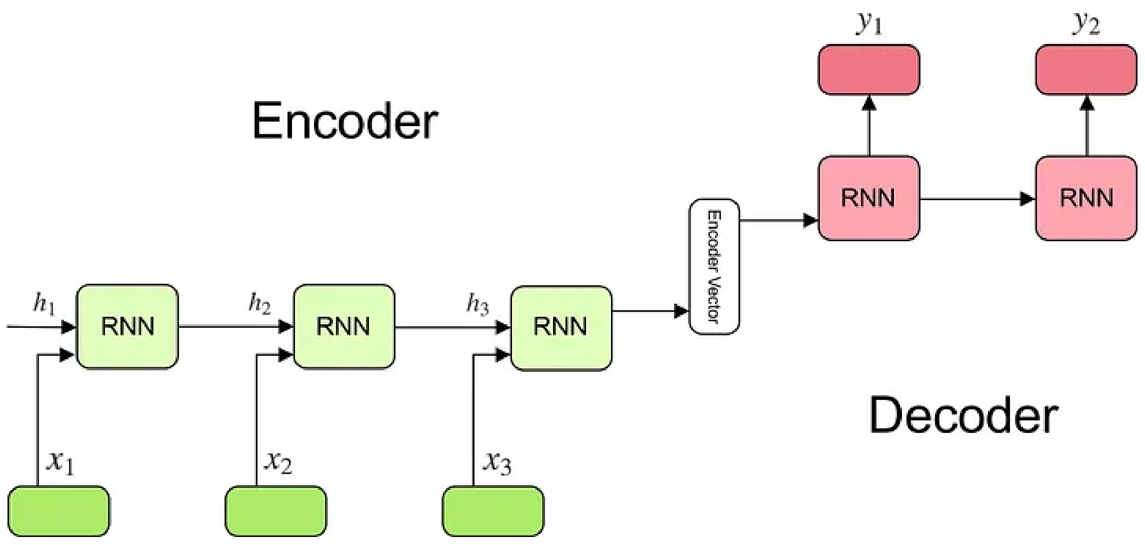


Figure 23: Encoder-decoder architecture. Data Source: Nadeem(2021, March 10) Encoder-Decoder, Sequence to Sequence architecture. https://medium.com/analytics-vidhya/encoders-decoders-sequence-to-sequence-architecture-5644efbb3392

The encoder is constructed by stacking multiple RNN cells (Figure 23). Each RNN cell takes 2 inputs. First is the output of previous RNN cell and second is a new token from the sequence of inputs. For a input sequence of size n, the RNN cells are repeated n times. Finally, the output of the last RNN cell which holds all the information for a particular image and partial text. The decoder predicts the next token based on this information, one at a time. The input to an encoder could be image, audio, or text depending on the problem at hand.

4.5 Recurrent Neural Network

In a normal classification problem, there are independent input values that get categorized into output classes. For e.g. classifying hand-written digits. These problems can be solved using a simple feed-forward neural network. In some use cases, the output data is a sequence of texts or time series information. For e.g. Stock value prediction, image captioning, etc. It is impossible and very unideal to solve these problems with a single feed-forward neural network.

The prediction for a time series data [26] is done one at a time. To predict the next token, it is important to have some information about the previous predictions. i.e. the neural network needs to have a memory of its own. A neural network with its own memory is a Recurrent Neural Network (Figure 24). RNN makes predictions from inputs and the output of the previous prediction.

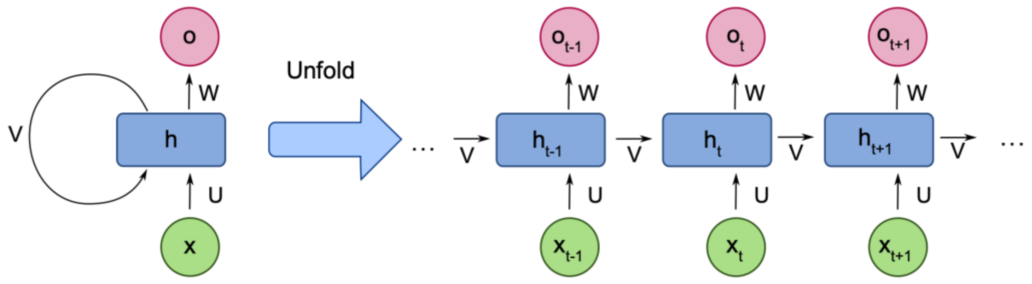


Figure 24: Recurrent Neural Network [fdeloche, CC BY-SA 4.0 via Wikimedia Commons]

The input to a layer is the previous state output combined with the current input state. Here is a mathematical representation of output at state t.

$$h_t = tanh\left(W_{hh} h_t - 1 + W_{xh} x_t\right) \tag{1}$$

W_{hh} - Weight of previous state

W_{xh} - Weight of current state

h_{t-1} – Output from previous state

x_t – Current state input

Although RNN can produce sequences of information, the major disadvantage is its memory. An RNN has a short memory, i.e. its output prediction depends only on the previous state. An RNN is also more prone to have a vanishing gradient problem. Both of these problems are overcome with an improvised version of an RNN called LSTM.

4.3 Long Short term memory (LSTM)

LSTM solves the short memory problem of an RNN model by adding a cell state. which is capable of learning long-term dependencies [27]. LSTM (Figure 25) models are called attention-based models because they pay selective attention to the output of the previous states and store the relevant information for future predictions. It is capable of finding the most relevant information from the input and deleting the irrelevant information. It has a lot of computation since it contains 4 units and each unit computes a different set of information. An LSTM model has 4 units - input gate, output gate, cell and forget gate.

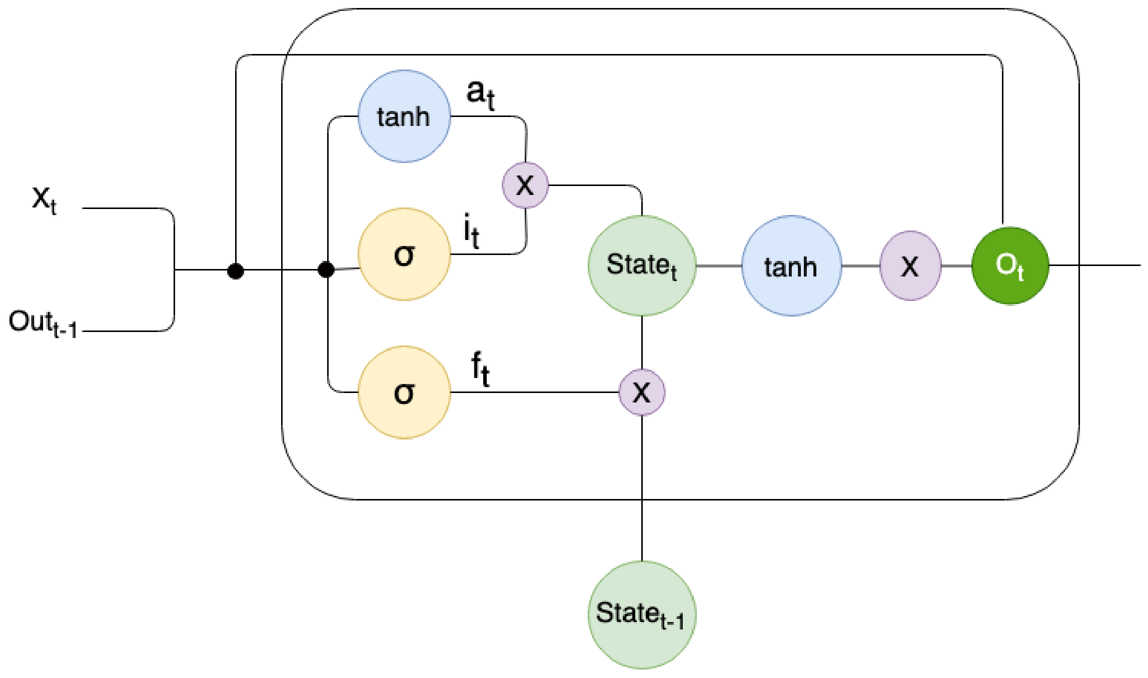


Figure 25: Long Short term memory (LSTM) architecture

Out of the four units in an LSTM layer, 3 are gates and one is a state. A State is where the model stores all the important information from the previous states. As shown in the Figure 26, state$_{t-1}$ and state are the state information at time t-1 and time t. The cell state information is updated after every step.

f$_t$ is the forget gate. forget gate helps in removing the information from the previous state. Forget gate takes has 2 inputs

x_t → input vector at time t

out_{t-1} → output vector from the previous time (t-1)

Dot products of these two inputs with their respective weights are added and passed through a sigmoid function to get the output f(t) which stores information that would help in removing information from the cell state.

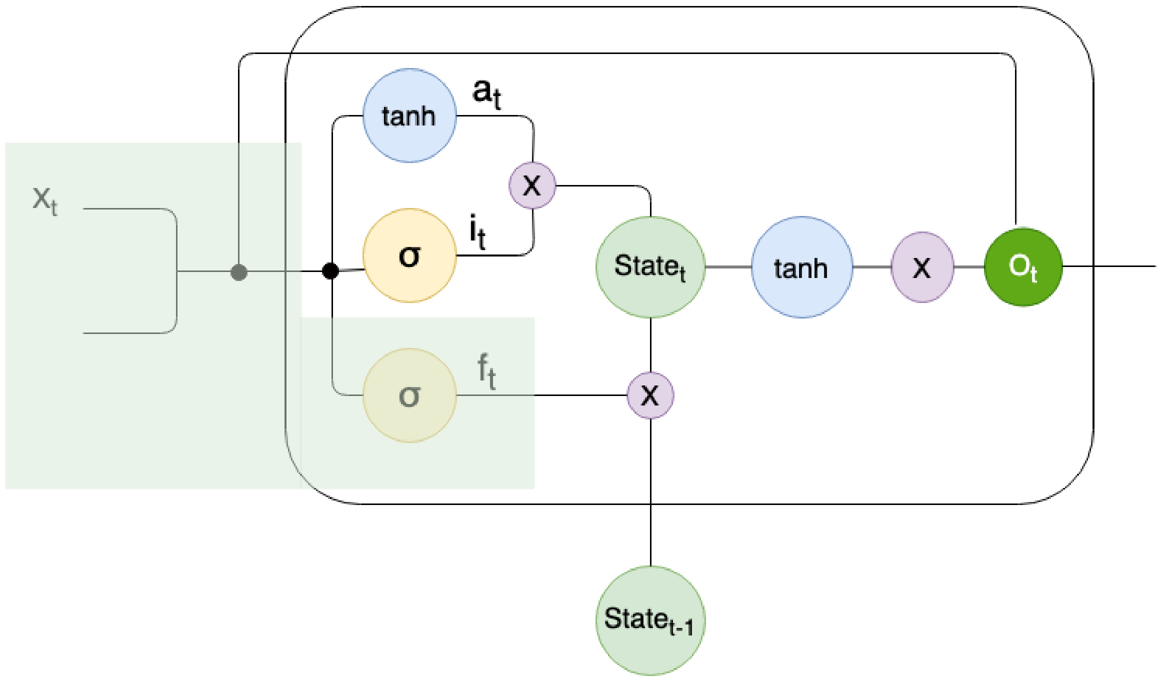


Figure 26: Forget gate in Long Short term memory(LSTM)

Equation 2 represents the mathematical representation of forget gate [28].

$$f_t = \sigma\left(W_t . \left[out_{t-1}, x_t\right] + b_f\right) \tag{2}$$

Apart from the information that needs to be forgotten, we also need to add new information to the cell state. This information is computed in 2 parts. The first part is the input gate. The input gates (Figure 27) determine the input values to update. The input gate takes the same 2 inputs as the forget gate and is passed through a sigmoid activation function.

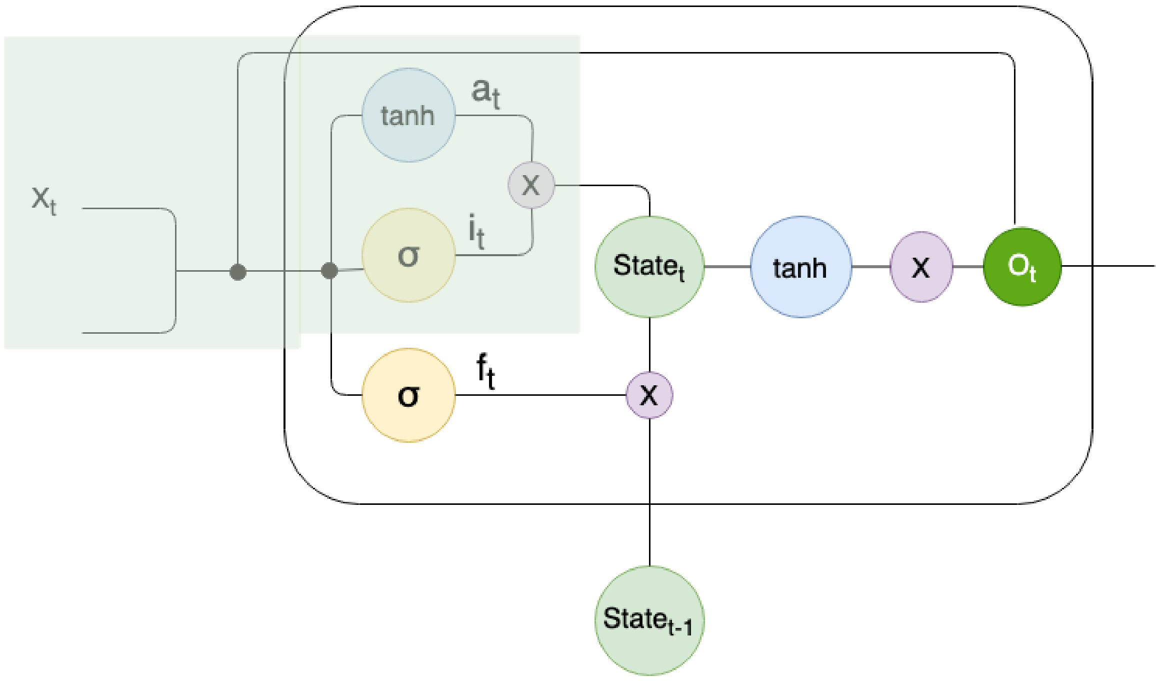


Figure 27: Input gate and tanh function in LSTM

$$i_t = \sigma\left(W_t . \left[out_{t-1}, x_t\right] + b_i\right) \tag{3}$$

The second part uses a different activation function tanh (Figure 27). This part outputs the new possible candidates of information that could be added to the cell state.

$$a_t = tanh(W_a.[out_{t-1}, x_t] + b_a) \tag{4}$$

Both the parts combined determine the final information to add to the cell state.

$$state_t = a_t * i_t + f_t * state_{t-1} \tag{5}$$

So far we have updated the cell state which holds the long-term memory information. The final step in the LSTM layer is to compute the output vector. The output vector is also a computer in two parts. The first part is the output gate which is also a sigmoid function of x_t and out_{t-1}. The second part is to pass the cell state through the tanh function which distributes the values between -1 and 1. Finally, the product of the output gate and the tanh function gives the final output.

$$O_t = \sigma(W_o.[out_{t-1}, x_t] + b_o) \tag{6}$$

$$out_t = O_t * tanh(state_t) \tag{7}$$

Hence an LSTM model uses the same layer to predict the sequence of time series data. The only difference from RNN is it maintains a cell state which gets updated at every step by a forget gate and input gates. Long Short term memory (LSTM) are designed to capture long term dependencies in sequential data. This long term information is stored in a memory cell which helps in capturing context and making more accurate predictions. LSTMs are often used in an encoder-decoder architecture as both an encoder (capturing input sequence) and a decoder (generating output sequence). An LSTM model is a practical choice for the problem at hand since it requires generation of sequential data from an input image.

CHAPTER 5

METHODOLOGY

The methodology for generating react code from an image design is greatly influenced by the research done by Tony Betranelli in pix2code [2]. The problem of generating a code from an image design is similar to image captioning. In both the cases, a textual description of the image provided is generated by a machine learning model. Similar to image captioning models, an encoder-decoder model was trained on the synthetic dataset generated in chapter 3.

5.1 High-level implementation

Graphical User Interface (GUI) is converted into a React App in 2 steps (Figure 28). The first step is to generate an intermediate code that holds the structure of the GUI along with tokens that represent each component type. The second step is to parse through the generated intermediate code and create a fresh React application.

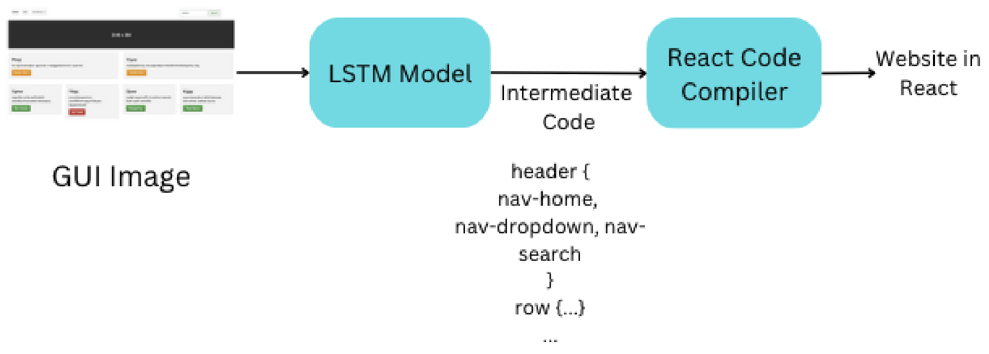
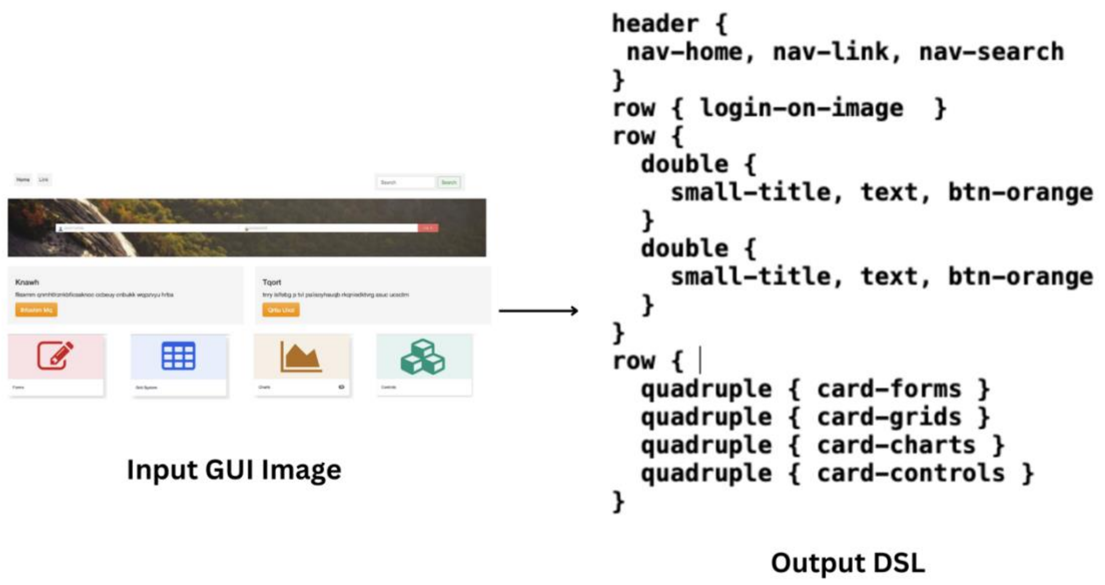


Figure 28: High level implementation.

The generating of intermediate code from Image design is implemented using an encoder-decoder architecture. This is explained in detail in later sections.



```
header {
 nav-home, nav-link, nav-search
}
row { login-on-image  }
row {
  double {
    small-title, text, btn-orange
  }
  double {
    small-title, text, btn-orange
  }
}
row { |
  quadruple { card-forms }
  quadruple { card-grids }
  quadruple { card-charts }
  quadruple { card-controls }
}
```

Figure 29: LSTM model input and output

The dataset (Figure 29) being referred to is similar to Image captioning data, which is a type of dataset commonly used in computer vision tasks where an algorithm is trained to generate textural descriptions of an input image.

5.2 Model Analysis

Generating textual code from an Image design is a problem similar to generating concise textual description from an Scene Image [20] . The standard approach to generating a textual information from a visual feature is using an encoder-decoder model [29]. Encoder-Decoder architecture have given very impressive results. Image captioning is one of the most popular use cases that showcase the capabilities of an Encoder-decoder model.

A Convolutional Neural Network (CNN) is used to encode image features whereas a Recurrent Neural Network (RNN) is used to encode textual descriptions. The 2 standard mechanism of implementing an encoder-decoder model is Inject model and merge model [21].

Inject Model

The traditional approach to generating textual information from an image uses a Convolution Neural Network to generate text that is relevant to the image. In Inject model (Figure 30), the recurrent neural network in the decoder uses both the image vector and text information as an input to generate the next token. Hence, compared to generating textual information entirely from an image, inject model uses partial textual information as a prefix.

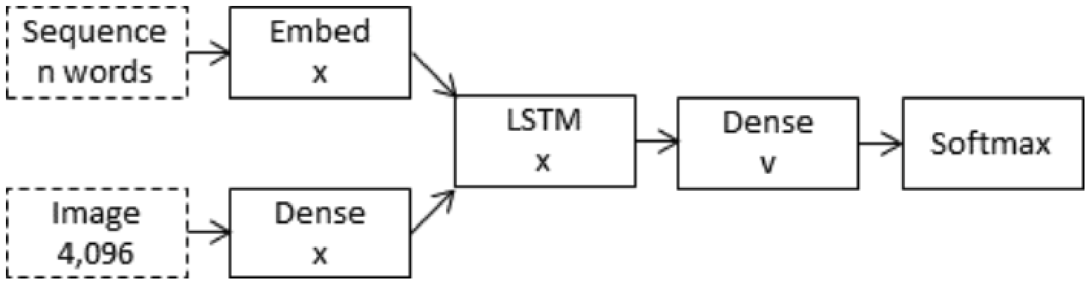


Figure 30: Inject Model. Data Source: Li, Y., Ouyang, W., Zhou, B., & Wang, K. (2017). Scene graph generation from objects, phrases and region captions. arXiv preprint arXiv:1708.02043.

Merge Model

In inject model, recurrent neural network only viewed only as a generator. Merge model uses CNN as an image encoder and RNN as a text encoder. Merging both image and textual feature (Figure 31) into a multimodal layer. Encoding the image and textual features separately makes sure that the RNN layer is not influenced by the CNN layer. Therefore, merge architecture maintains the features separately, merging them at the later stage.

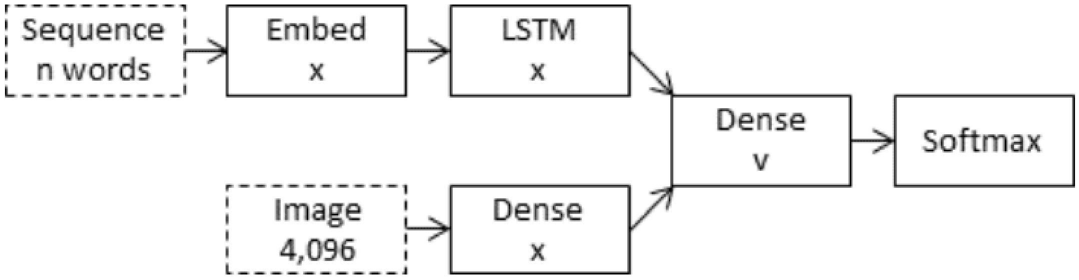


Figure 31: Merge model. Data Source: Data Source: Li, Y., Ouyang, W., Zhou, B., & Wang, K. (2017). Scene graph generation from objects, phrases and region captions. arXiv preprint arXiv:1708.02043.

In Conclusion, the merge architecture combines RNN encoded text with the visual features encoded separately to predict the next token, whereas Inject model encode image using a CNN and prefix words as an input to an RNN generator. Inject model handles more number of parameters since it is embedding text and image both with an RNN generator.

5.3 Encoder-decoder architecture using inject model

The architecture used in the proposed implementation is an inject model (Figure 32). The reason for choosing inject model over merge model is because the image and prefix in the problem at hand are tightly related. The image design holds the details of the objects present, their types and the structure. Hence the next token to be predicted highly depends on the caption and the image features both.

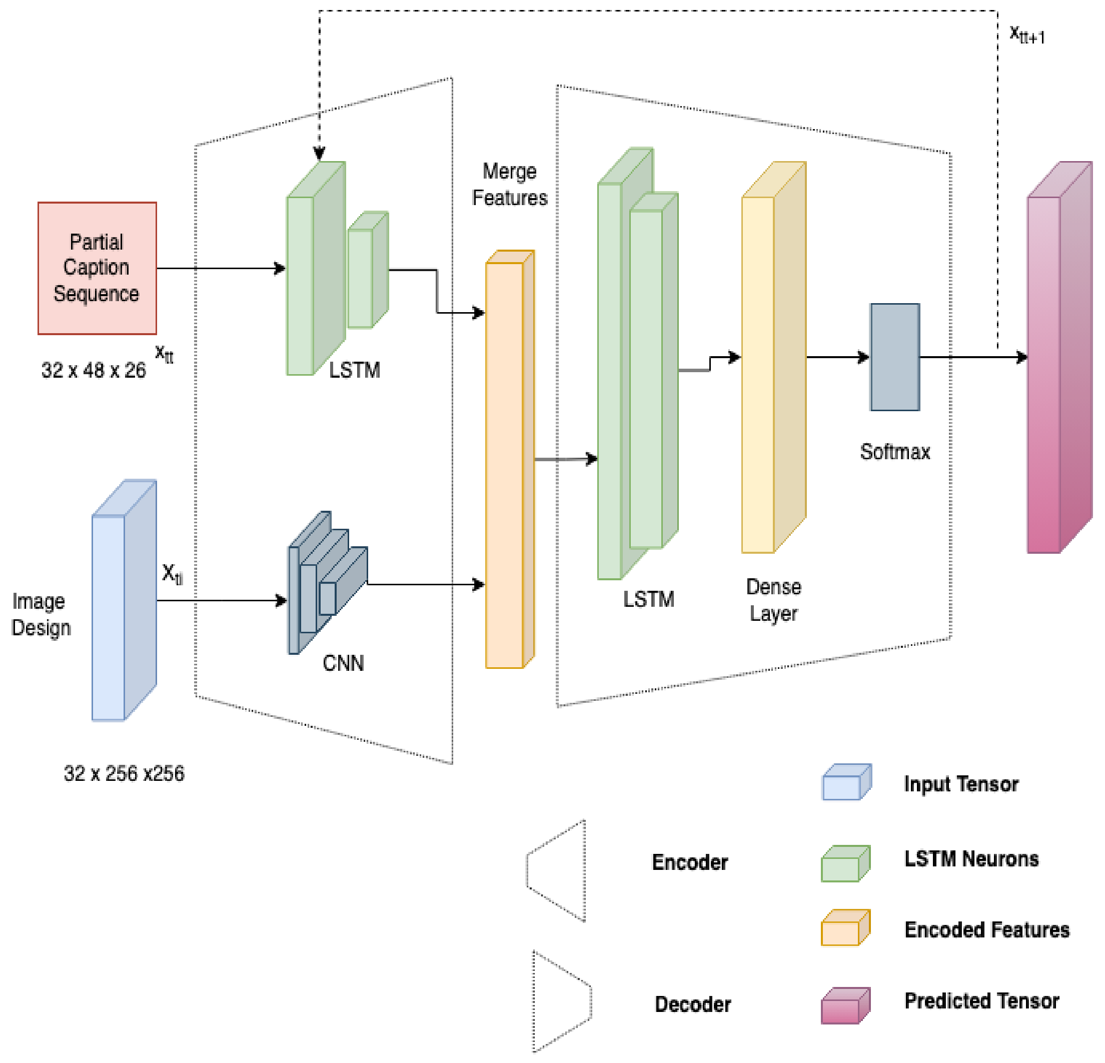


Figure 32: Overview of the encoder-decoder architecture used to generate Domain Specific language (DSL) from an image design.

Vision Model

The input image is passed through a sequence of Convolutional Neural networks. Based on empirical results, the number of filters used for each CNN is 64 or 128. Each convolution layer is followed by a pooling layer of (2,2) size. A dropout layer is added at the end of each convolutional neural network (CNN). Dropping few nodes in a neural network helps to reduce the overfitting problem. A dropout of 0.25-0.3 was used as the final layer of each network. The convolutional features are finally flattened and passed through a couple of dense layers.

Language Model

The Domain Specific language contains a sequence of tokens which corresponds to the layout of the web GUI along with the identification of the individual elements. These sequence of tokens are used as the input to the language model. These sequence of tokens are of fixed length. Based on some empirical results, both 48 and 60 gave similar results. The vocab size of the synthetic data created is 26. Therefore, the tokens are converted into one-hot encoded vectors of size 26 for each token. Language model is a stack of Long Short term model (LSTM) layers [2]. The image features from the Convolution neural network and the encoded sequence of tokens are concatenated together and passed as an input to the decoder. The decoder is also a stack of LSTM layers followed Dense layer using a SoftMax activation function.

5.4 Data pre-processing

Dataset contains 3500 images of the GUI interface with the corresponding DSL intermediate code. We do an 80-20 split for training and testing. To reduce the training time, the input image is converted and saved into a NumPy array. While converting images into NumPy arrays, the image is compressed to reduce the input dimension. For an LSTM model, it is important to determine the length of the input sequence. While training the input sequence length is important so that the model trains on all different states of inputs. Based on the empirical experiment done by pix2code, the length of the sequence taken was

48. This means each data would generate 48 samples. Hence the total training sample size is 135k. For each data, the output is prepended with 48 dummy empty strings. The input to the model is the image and the partial caption. The output is the next expected token. The data sample for each data point is created with a sliding window algorithm where the output token is appended to the input partial caption to predict the next token. This process of predicting the next token continues until the model predicts the end token.

To be able to terminate the LSTM model, we need a start and an end token. These two tokens are prepended and appended to the sequence of tokens in the output.

5.5 Training

The LSTM model has 2 inputs - a numpy image array and a one hot encoded sequence of partial code. The size of the partial code used was 48 based on empirical results and as suggested in pix2code [2] paper.

The dataset size of 3500 Images contributes training samples of 178000. Each image/DSL data contributes samples size of length of tokens in DSL – 48. The input (image, caption) and the output (next token) is generated using a sliding window algorithm (Figure 31). The slice from i to i+48 from sequence list is taken as an input and the token at i + 48th is mapped as an output token. The same image numpy array is used as an input for all the samples of a particular data.

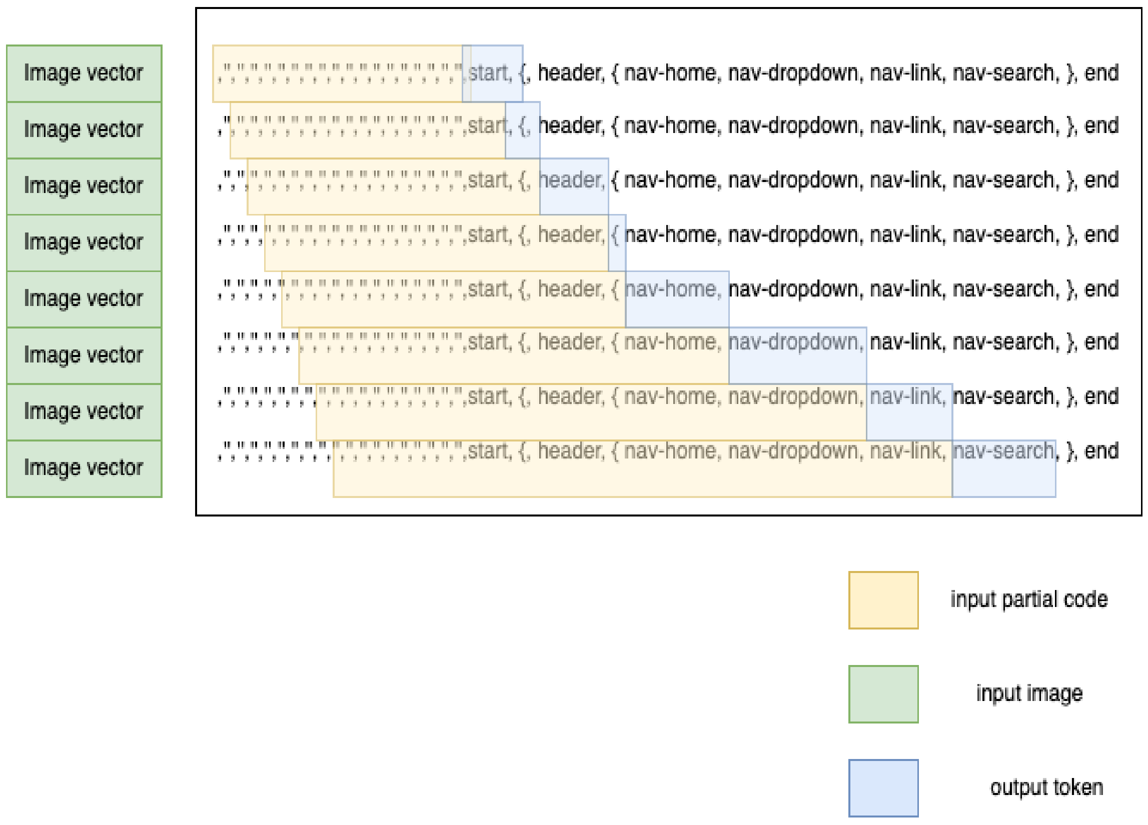


Image vector
,'','','','','','','','','','','','','','','','','','','',start, {, header, { nav-home, nav-dropdown, nav-link, nav-search, }, end
input partial code
input image
output token

Figure 33: dataset for encoder-decoder model.

The sequence of tokens is padded with 48 empty strings so that the initial tokens could also be used as an output token. If a Domain Specific language of a particular design image is of length 120. That image contributes 120 samples using the sliding window technique.

The LSTM model takes two Sequential models (Figure 34) as input. Both sequential models are merged together to create a feature vector.

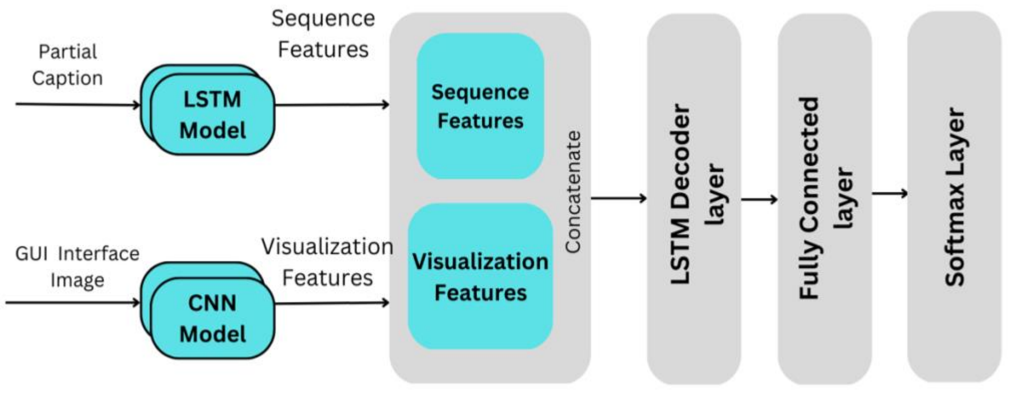
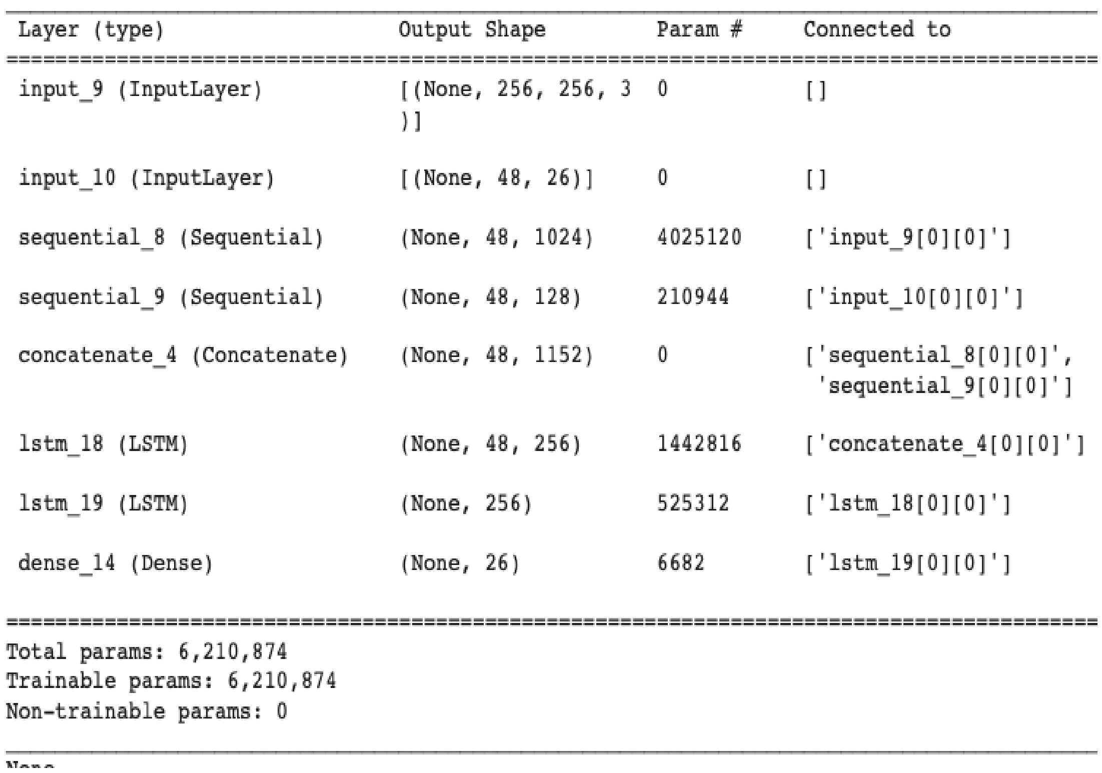


Figure 34: High level design of LSTM model.

The visual features from the CNN model and the Sequence features from the LSTM Model are concatenated together to form a complete feature vector. These feature vectors are passed to another LSTM layer which functions as a decoder. The decoder has a 2 LSTM layers followed by another dense layer. SoftMax activation function is used in the last dense layer since the model is essentially a classification model and SoftMax is the most preferred activation function for classification models.

The model was trained over 15 epochs. Each CNN layer is followed with a pooling and a dropout layer [32]. This helps in reducing the number of parameters the model has to learn on and regularize the overfitting of the neural network. The training was done at a batch size of 32. The summary of the layers of the implemented model is represented in Figure 33. The 2 sequential models are the vision model and the language model (Figure 35). The output of the last dense layer is 26 which corresponds to the vocab size. The image is downscaled to a size of 256x256. The image has 3 channels for RGB. Hence the size of input_1 is 256x256x3.



```
_______________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
=======================================================================================
 input_9 (InputLayer)           [(None, 256, 256, 3  0           []
                                )]

 input_10 (InputLayer)          [(None, 48, 26)]     0           []

 sequential_8 (Sequential)      (None, 48, 1024)     4025120     ['input_9[0][0]']

 sequential_9 (Sequential)      (None, 48, 128)      210944      ['input_10[0][0]']

 concatenate_4 (Concatenate)    (None, 48, 1152)     0           ['sequential_8[0][0]',
                                                                  'sequential_9[0][0]']

 lstm_18 (LSTM)                 (None, 48, 256)      1442816     ['concatenate_4[0][0]']

 lstm_19 (LSTM)                 (None, 256)          525312      ['lstm_18[0][0]']

 dense_14 (Dense)               (None, 26)           6682        ['lstm_19[0][0]']

=======================================================================================
Total params: 6,210,874
Trainable params: 6,210,874
Non-trainable params: 0
_______________________________________________________________________________________
```

Figure 35: Model summary

5.6 Compiler

The compiler takes in an intermediate code (DSL) as input and transforms it into a React code (Figure 34). Each token in the Domain specific language maps to a React component class. All the corresponding react component class [30] are prebuilt with default configurations. For e.g. Token nav-search maps to a React class NavSearch.js. The opening and closing curly braces are mapped to Container classes. Compiler takes the intermediate Domain Specific Language (DSL) i.e. the sequence of tokens and transforms it into a JavaScript React code (Figure 36).

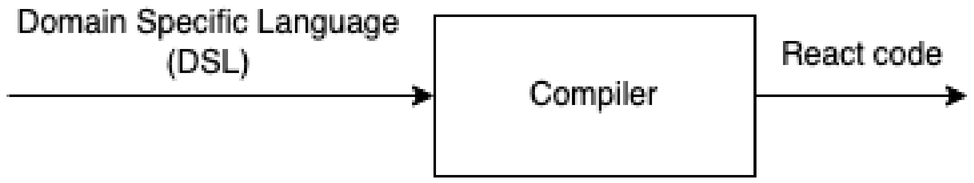
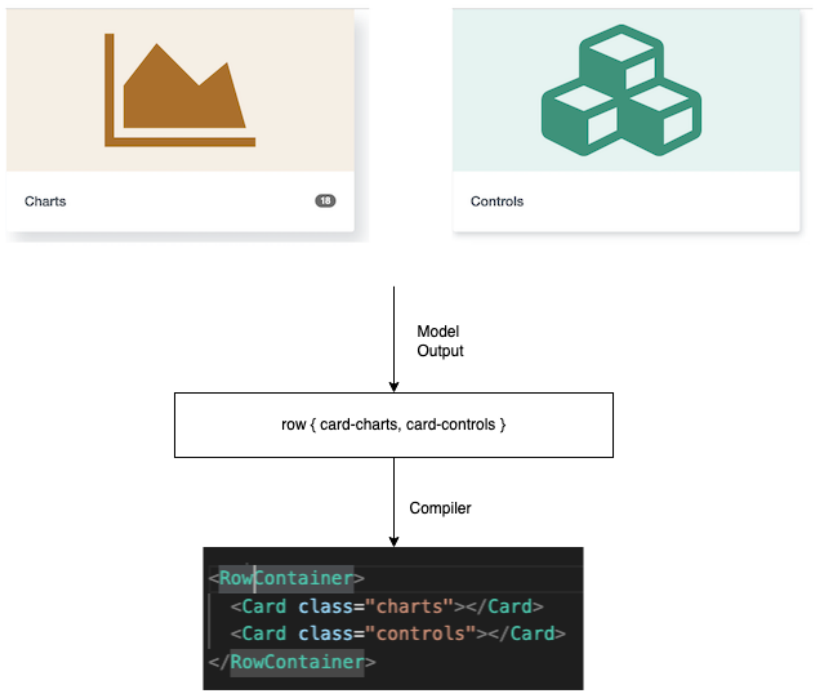


Figure 36: Compiler

The Domain Specific Language (DSL) used is synthetically generated and is very lightweight. The DSL follows a row first layout. All the horizontally aligned components are stored inside a row tag (Figure 37). Components without a tag are assumed to be vertically aligned.



```
<RowContainer>
  <Card class="charts"></Card>
  <Card class="controls"></Card>
</RowContainer>
```

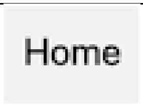
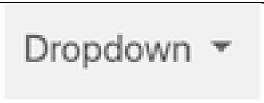
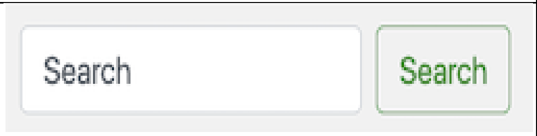
Figure 37: Compiler Input and output for a row container

There are 2 types of tokens. The first type of tokens are the ones that stores layout information (Table 1) and the second type of tokens correspond to a React class (Table 2).

Token	Meaning
row	All the components inside are horizontally aligned
header	Navigation bar
double	2 elements take equal width horizontally
single	1 element take the entire width
quadraple	4 components take equal width horizontally
{ placeholder}	Components inside an opening and closing braces are vertically aligned

Table 1: tokens that store layout

The class labels/tokens in Table 1 are used to store the layout of elements in an image design. Each curly braces is prefixed by a token. This token represents whether the components inside are horizontally aligned or vertically aligned. There is a special token for header which by default is horizontally aligned.

Token	React class	Component Image
nav-home	<NavButton>	
nav-dropdown	<NavDropdown>	
nav-search	<NavSearch>	

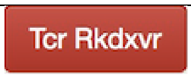
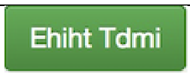
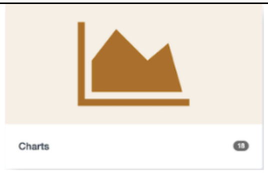
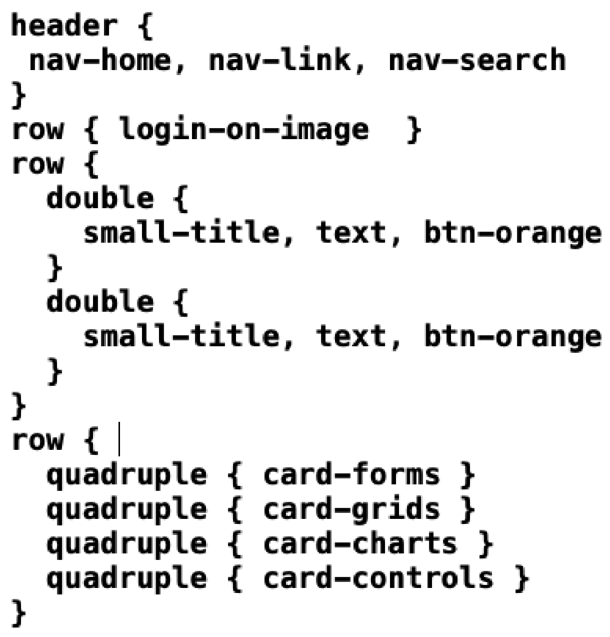
Btn-red	<Button class=”red”>	
Btn-green	<Button class=”green”>	
card-charts	<Card class=”charts”>	

Table 2: tokens that corresponding to components

Some of the tokens that directly correspond to a React component a listed in Table 2. Most of these tokens are directly mapped to a React components. Some of the components like buttons and cards have an additional class associated since except the color or icons, the components have same structure.

As represented in Figure 38, the intermediate code is essentially a sequence of tokens. Each token separated by space or a new line has its own meaning (Table 1 and Table 2). The token either signifies a layout or a particular HTML element. The compiler is a recursive function that takes in the sequence of tokens and recalls itself for each token that represents the layout until it reaches a leaf node.



```
header {
 nav-home, nav-link, nav-search
}
row { login-on-image  }
row {
  double {
    small-title, text, btn-orange
  }
  double {
    small-title, text, btn-orange
  }
}
row {
  quadruple { card-forms }
  quadruple { card-grids }
  quadruple { card-charts }
  quadruple { card-controls }
}
```

Figure 38: Input to the compiler

A leaf node is the token that represents the actual HTML element. One of the issues to generate a React code is it has a starting and an ending tag. The compiler was implemented to accommodate this. An example output of the compiler is shown in Listing 1.

```
<RowContainer className="row-wrapper">
   <NavHeader>
     <NavButton>Home</NavButton>
     <NavButton>Link</NavButton>
     <NavSearch></NavSearch>
   </NavHeader>
   <RowContainer className="row-wrapper">
     <RowContainer className="flex-row">
       <RowContainer className="flex-column">
         <LoginOnImage></LoginOnImage>
       </RowContainer>
```

```
</RowContainer>
<RowContainer className="row-wrapper">
 <RowContainer className="flex-row">
  <RowContainer className="flex-row">
   <RowContainer className="flex-column">
    <SmallTitle>Ddsfaybhd</SmallTitle>
    <Text>fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd</Text>
    <Button class="orange">uiedncids</Button>
   </RowContainer>
  </RowContainer>
  <RowContainer className="flex-row">
   <RowContainer className="flex-column">
    <SmallTitle>Ddsfaybhd</SmallTitle>
    <Text>fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd</Text>
    <Button class="orange">uiedncids</Button>
   </RowContainer>
  </RowContainer>
 </RowContainer>
 <RowContainer className="row-wrapper">
  <RowContainer className="flex-row">
   <RowContainer className="flex-row">
    <RowContainer className="flex-column">
     <Card class="forms"></Card>
    </RowContainer>
   </RowContainer>
   <RowContainer className="flex-row">
    <RowContainer className="flex-column">
     <Card class="grids"></Card>
    </RowContainer>
   </RowContainer>
   <RowContainer className="flex-row">
    <RowContainer className="flex-column">
     <Card class="charts"></Card>
    </RowContainer>
   </RowContainer>
   <RowContainer className="flex-row">
```

```
            <RowContainer className="flex-column">
             <Card class="controls"></Card>
            </RowContainer>
           </RowContainer>
          </RowContainer>
         </RowContainer>
        </RowContainer>
       </RowContainer>
      </RowContainer>
```

Listing 1: Compiler output react code

The domain specific code has curly braces as a token that holds the nested structure of the elements in the GUI Image design. These curly braces gets transformed into Row Container with start and end tags. The prefix token before the curly braces determines the styling applied to the row container. The token row gets transformed into Row Container with flex-row as the styling class. The text used in components like Small title and text is a dummy text. The focus of this thesis was to generate the layout with correct identification of the components which was achieved. The compiler creates same component for similar tokens like buttons and card. The different buttons and class are distinguished by the class style added to those components. The limitation of the implemented compiler is that it works only for a single page application and generates the structure as a single JavaScript file. For more complex designs, the compiler architecture could be improved to generate smaller react components to generate a more manageable and easily editable code.

CHAPTER 6

RESULTS

6.1 LSTM Model evaluation

A recurrent neural network is essentially a classification model that feeds the output back as an input. Figure 39 shows the distribution of class labels in the Domain Specific Language (DSL) corresponding to a set 200 image designs used in the validation dataset.

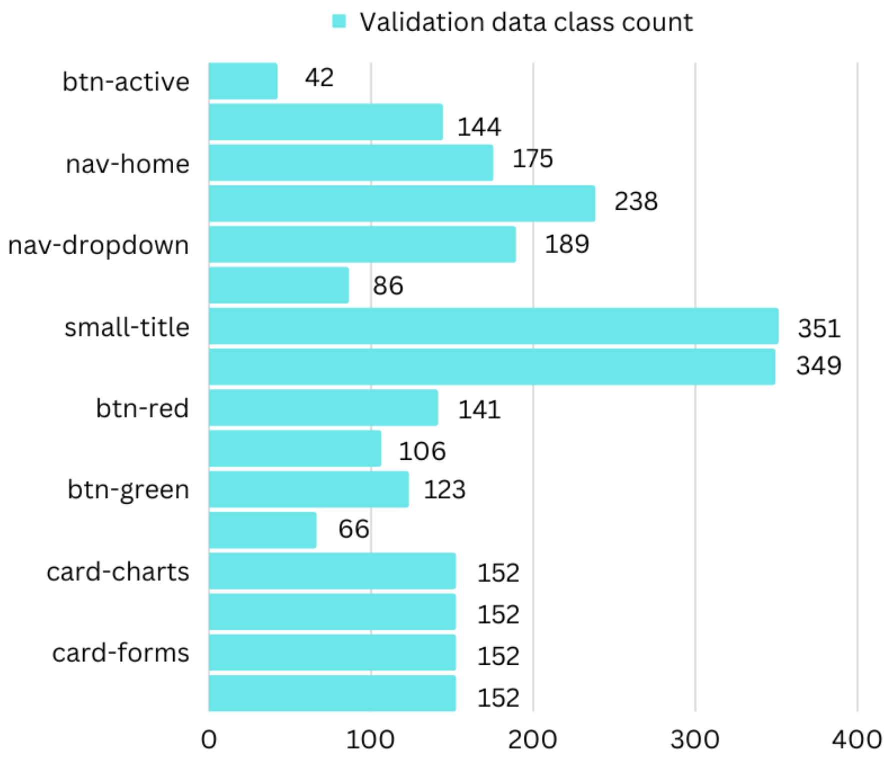


Figure 39: Frequency of class labels in validation data

There were two contributions in this research. The first one is an enhanced dataset which had 3500 image designs with corresponding Intermediate Domain Specific Code (DSL). A pipeline was proposed to generate more synthetic dataset. The other part of the research was using the generated dataset to train an LSTM model. This trained model had an accuracy of 96.7%. Feature extraction from the images was done without using any pretrained models. The CNN model was trained with the new generated dataset. In this research, we have used greedy algorithm for token prediction. It could be optimized further with beam search [31] and argmax methods.

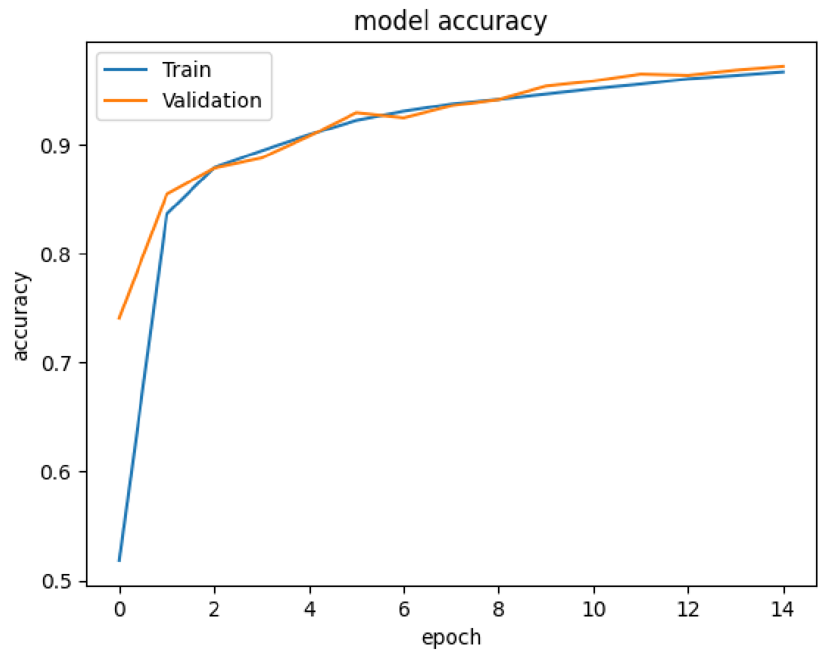


Figure 40: Accuracy after every epoch

Validation accuracy (Figure 40) is slightly higher in every epoch because the model has few dropout layers in the Convolutional Neural Network layers. This dropout layers drops few key features while training which is not the case during validation. After 15 epochs the accuracy of the model was 96%.

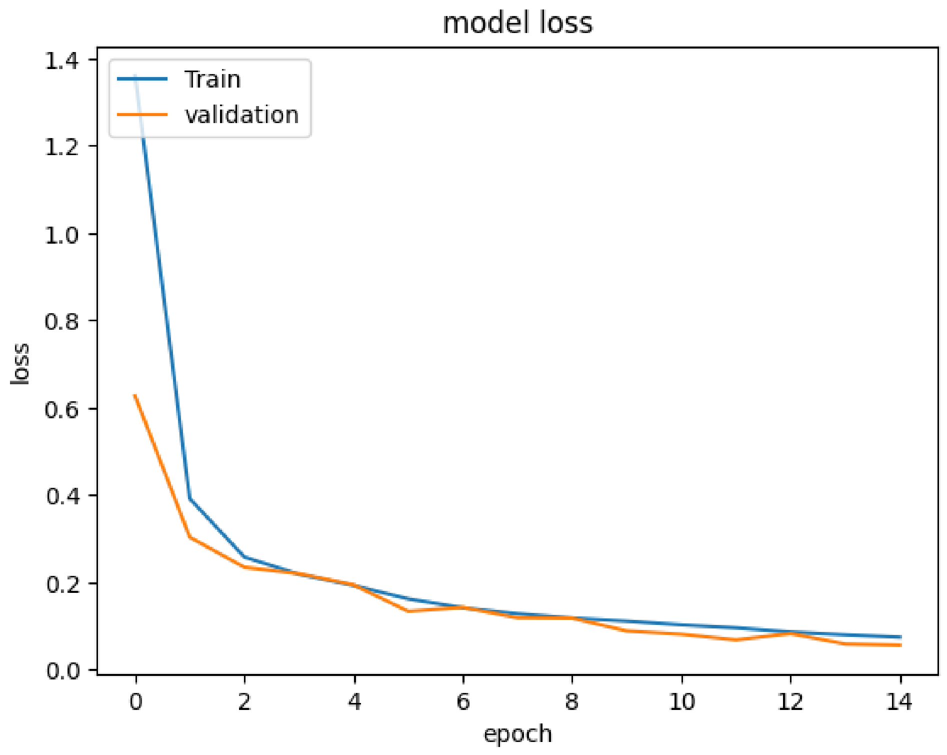


Figure 41: Training loss after every epoch

From epoch 12 to 15, the loss reached its global minima (Figure 41). The first few epochs indicate that the model trained very quickly. This is because some of the class labels are easier to learn like headers, Containers, home button etc. Other classes like various types of buttons, cards are the ones that required more training epochs.

The model has a Receiver Operating Character area under curve (ROC AUC) of 0.96 (Figure 42). Accuracy signifies what percentage of the predictions were accurate but ignores the probabilities of the output layer. An ROC AUC curves signifies the confidence of a Neural Network model in distinguishing between classes.

The mean Average Accuracy (mAP) of individual class varies from 87-100%. The dataset used for training is synthetic and lightweight. The model can distinguish all the elements with good accuracy except the different buttons. This is probably because buttons have similar shape and size in the synthetic data.

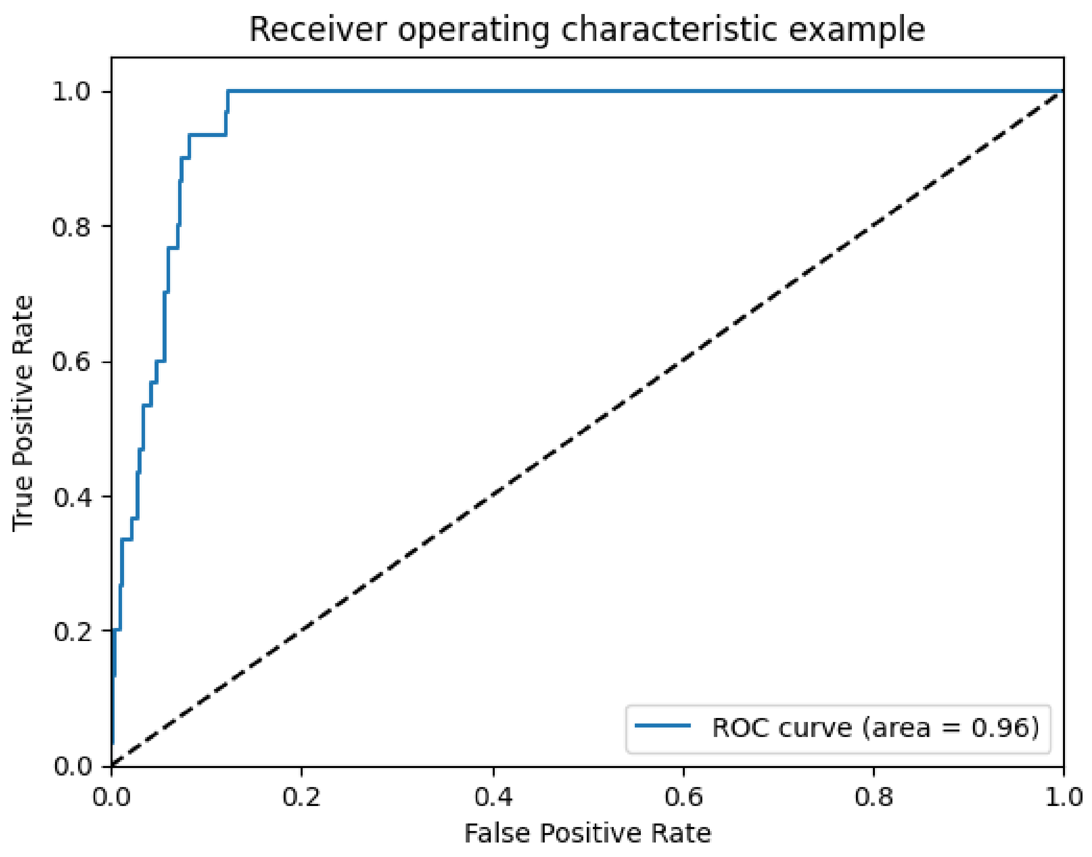


Figure 42: micro average ROC AUC Curve

The only difference is the background colour. Some of the elements like nav-home, container text, carousels have a very high accuracy because they occur more frequently in the synthetic dataset. There are also some limitations in the variations in some of the elements like Navigation home.

Determining the mean average precision (mAP) for individual class elements is an essential metric in determining the performance of the model. Table 3 shows the performance of the model in terms of how accurate it is in classifying various types of elements.

Element	Ground-truth	True Positive	False Positive	mAP (%)
Btn-active	42	40	4	95
Btn-inactive	144	141	7	97.6
Nav-home	175	175	0	100
Nav-link	238	213	10	90
Nav-dropdown	189	179	21	95
Nav-search	86	86	4	100
Container	1605	1605	0	100
Small-title	351	349	5	99
text	349	349	0	100
Btn-red	141	123	0	87
Btn-orange	106	95	14	90
Btn-green	123	118	27	93
carousel	66	66	0	100
Card-charts	152	151	0	99
Card-grids	152	151	2	99
Card-forms	152	145	0	95
Card-controls	152	151	7	99

Table 3: mAP (%) for each output class

Precision, recall and f1 score (Table 4) are measured for a validation dataset size of 200 image designs. Components like nav-home, small-title, text, carousel gave a very high accuracy. This is potentially because each of these elements had an evident distinctive property. For e.g., carousels take entire width with 4 dotted items. On the other hand, elements like buttons had a low accuracy since geometrically all the buttons had same features and the only distinguishing factor was the background color.

Element	Precision	Recall	F1 Score
Btn-active	0.98	0.96	0.95
Btn-inactive	0.97	0.95	0.96
Nav-home	1	1	1
Nav-link	0.95	0.89	0.92
Nav-dropdown	0.89	0.94	0.92
Nav-search	0.95	0.98	0.97
Small-title	0.99	1	0.99
text	0.99	1	0.99
Btn-red	0.87	0.82	0.84
Btn-orange	0.90	0.92	0.90
Btn-green	0.94	0.97	0.95
carousel	1	1	1
Card-charts	0.99	0.98	0.98

Card-grids	0.97	0.98	0.98
Card-forms	0.96	0.95	0.97
Card-controls	0.95	0.99	0.97

Table 4: Precision, Recall and f1score for each output class

The inspiration of this topic was pix2code [2]. The methodology proposed is greatly influenced by pix2code implementation with enhanced dataset. Pix2code dataset had 19 labels with only few html elements. In this thesis, the enhanced dataset has 26 labels with more complication elements like cards, login forms, icons, carousel etc. General comparison between the results of implemented approach and pix2code results (Table 4) is listed.

	Pix2code	Pics2React
Vocab size	19	26
Accuracy	0.99	0.96
loss	0.02	0.07
Compiles to	Pure Html CSS	React Application

Table 5: Comparing pix2code results with current results.

6.2 Visual Comparison

Comparing the input image and the generated website visually is a crucial factor in evaluation the task at hand. To facilitate this comparison, three example of GUI Image designs were selected. Each example includes two images: the first one marked as (A) depicts the ground truth of GUI image design that was used as an input, and the second one shows a snapshot of the actual results of a functional website generated using the proposed methodology.

Example I

In this example, the selected GUI Image design include login forms, cards with icons, dropdowns, and buttons. The header elements, including navigation home, dropdown and link were accurately predicted, except for the orange button, which was predicted as green. The other elements like login form and cards were correctly predicted.

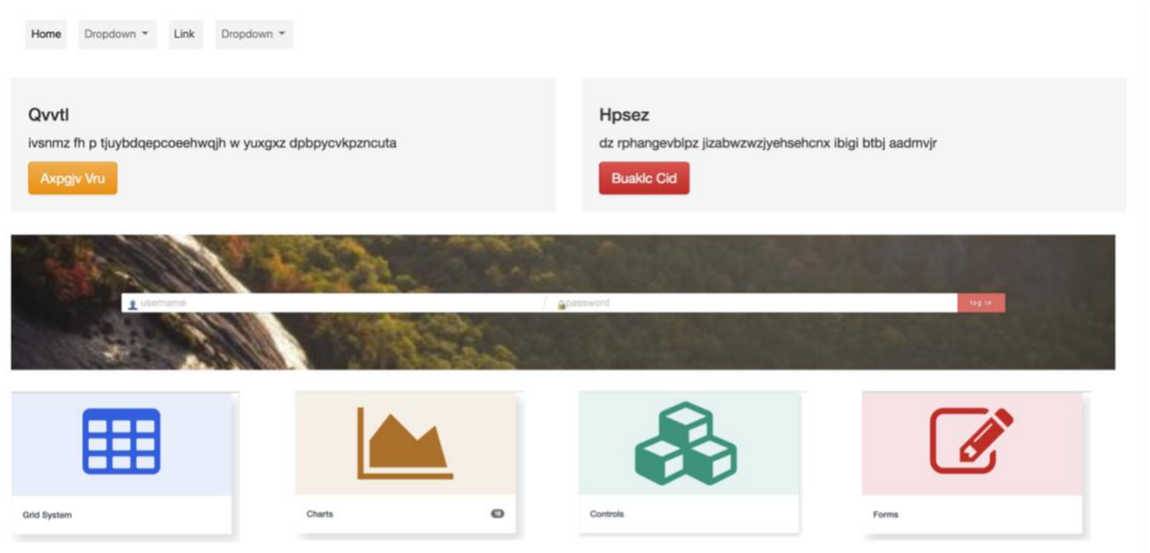

Home
Dropdown
Link
Dropdown
Qvvtl
ivsnmz fh p tjuybdqepcoeehwqjh w yuxgxz dpbpycvkpzncuta
Axpgjv Vru
Hpsez
dz rphangevblpz jizabwzwzjyehsehcnx ibigi btbj aadmvjr
Buaklc Cid
username
password
log in
Grid System
Charts
Controls
Forms

Figure 43(A) Ground truth of GUI Image design.

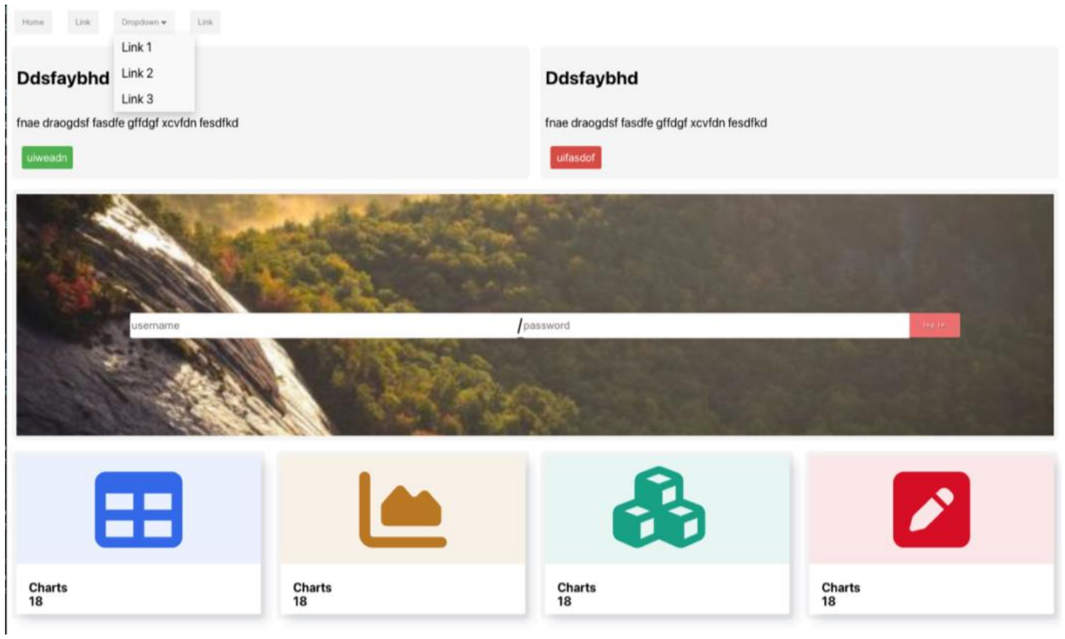


Figure 43(B) Actual Result for GUI Image design.

Example II

In this example, the input image design includes several elements such as a search bar, carousel, cards with icons, and buttons. The carousels and search bar were accurately predicted, demonstrating the proposed methodology's capability to locate and identify various HTML elements.

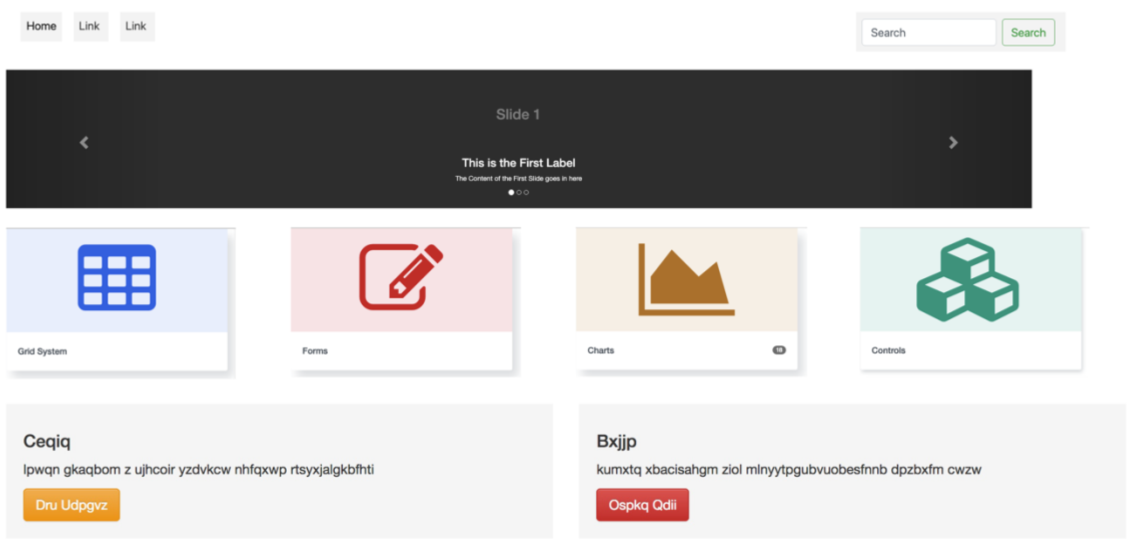

Home
Link
Link
Search
Search
Slide 1
This is the First Label
The Content of the First Slide goes in here
Grid System
Forms
Charts
Controls
Ceqiq
lpwqn gkaqbom z ujhcoir yzdvkcw nhfqxwp rtsyxjalgkbfhti
Dru Udpgvz
Bxjjp
kumxtq xbacisahgm ziol mlnyytpgubvuobesfnnb dpzbxfm cwzw
Ospkq Qdii

Figure 44 (A) Ground truth of GUI Image design.

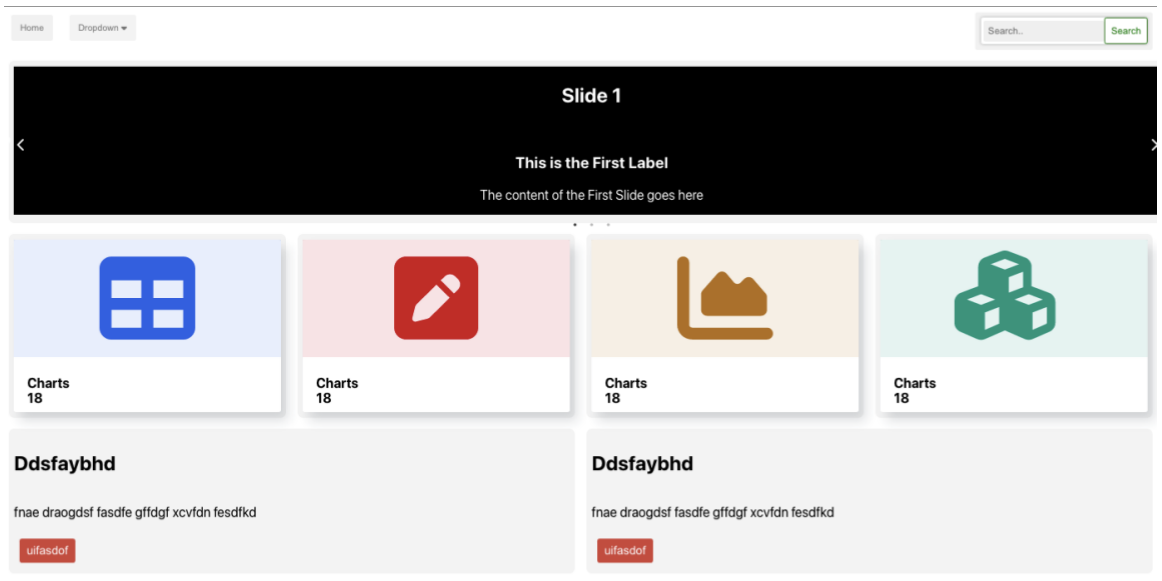
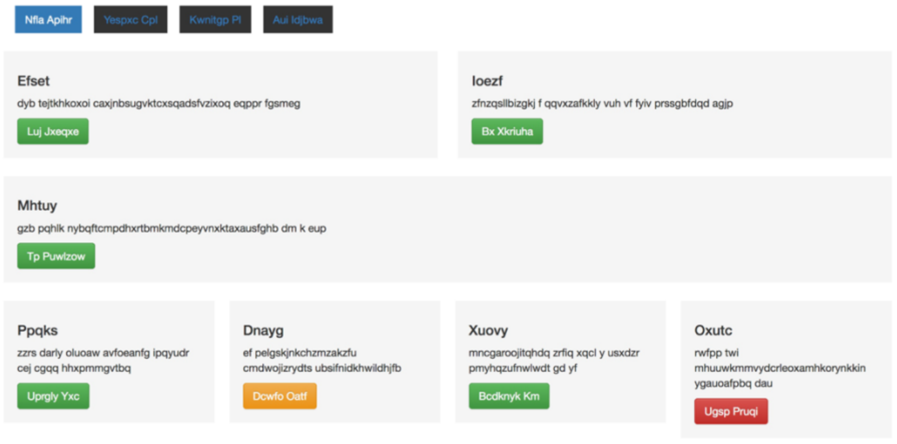

Home
Dropdown
Search..
Search
Slide 1
This is the First Label
The content of the First Slide goes here
Charts
18
Charts
18
Charts
18
Charts
18
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uifasdof
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uifasdof

Figure 44 (B) Actual Results of GUI Image design.

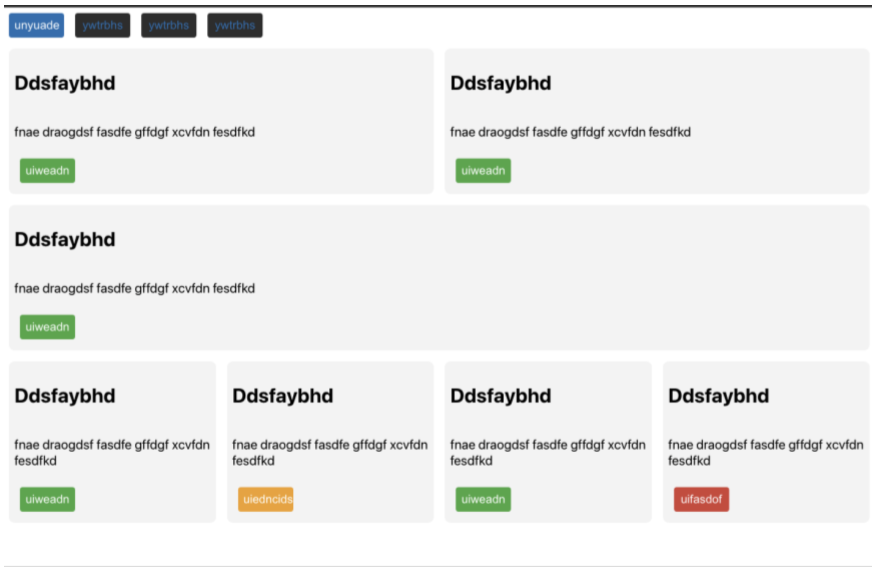
Example III

Another example taken to examine the model's capability to distinguish between different types of buttons. The model gave positive results and detected all variations of the button accurately. These findings indicate that the model can distinguish between different types of buttons even if the shape and size are same, except for a different background colour.



Nfla Apihr
Yespxc Cpl
Kwnitgp Pl
Aui Idjbwa
Efset
dyb tejtkhkoxoi caxjnbsugvktcxsqadsfvzixoq eqppr fgsmeg
Luj Jxeqxe
Ioezf
zfnzqsllbizgkj f qqvxzafkkly vuh vf fyiv prssgbfdqd agjp
Bx Xkriuha
Mhtuy
gzb pqhlk nybqftcmpdhxrtbmkmdcpeyvnxktaxausfghb dm k eup
Tp Puwlzow
Ppqks
zzrs darly oluoaw avfoeanfg ipqyudr cej cgqq hhxpmmgvtbq
Uprgly Yxc
Dnayg
ef pelgskjnkchzmzakzfu cmdwojizrydts ubsifnidkhwildhjfb
Dcwfo Oatf
Xuovy
mncgaroojitqhdq zrfiq xqcl y usxdzr pmyhqzufnwlwdt gd yf
Bcdknyk Km
Oxutc
rwfpp twi mhuuwkmmvydcrleoxamhkorynkkin ygauoafpbq dau
Ugsp Pruqi

Figure 45 (A) Ground Truth of GUI Image.



unyuade
ywtrbhs
ywtrbhs
ywtrbhs
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiweadn
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiweadn
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiweadn
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiweadn
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiedncids
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uiweadn
Ddsfaybhd
fnae draogdsf fasdfe gffdgf xcvfdn fesdfkd
uifasdof

Figure 45 (B) Actual Results of GUI Image design.

CHAPTER 7

CONCLUSION

Generating a low-level frontend code from an Image design is a challenging problem that is further complicated by the fact that low-level frontend code can vary across different frameworks. To address this challenge, an intermediate code that represents the image design is used which makes the implementation framework agnostic. This intermediate code can then be compiled into any framework of choice. However, availability of a good dataset is a prominent obstacle to automate this task. Fortunately, synthetic datasets can be generated for this purpose. This thesis utilized a relatively small synthetically generated dataset, and a pipeline was proposed for generating more complex dataset. This pipeline requires minimal manual intervention is required for adding more complex elements with the proposed data generation pipeline. However, one drawbacks of the proposed solution are that it doesn't extract pixel perfect details from the design image, resulting in some differences in height, width, and positions. Nonetheless, the project aims to give the developer a very good starting point of the website which requires minimal tweaks. This approach would work well with libraries like bootstrap which has a good database of component libraries.

CHAPTER 8

FUTURE WORK

The project focuses on both generating new synthetic dataset and using it to automate the generation of web GUI. The dataset generated using this pipeline has limited number of elements. A more complicated dataset could be generated and trained using the proposed methodology. The current dataset focuses on basic website layout. A new dataset with nested structures which has more html elements like checkboxes, radio buttons and different themes could be generated.

The LSTM model follows a greedy approach to make predictions. i.e., it selects the class with maximum probability at each time step. This could be improved using a beam search which keeps record of multiple possible classes and picks the most relevant one based on future predictions.

In this thesis, the algorithm focuses on implementing a single page application which generates a react app from scratch. This could be improved by adding GUI generation for multiple pages and linking them together. Linking multiple pages together is a challenging task. One possible solution is to use a dataset with folder structures and filenames which could be used in the compiler to link various pages into a single react application. Linking between pages could be done by following a folder naming convention where the folder is named with the token used in the intermediate code. For multiple occurrences of the same token, the folder name could be appended with a sequence number.

REFERENCES

[1] Benjamin Wilkins (2017, September 7) Sketching Interfaces – Generating code from low fidelity wireframes. https://airbnb.design/sketching-interfaces/

[2] Tony Beltramelli. 2018. Pix2code: Generating Code from a Graphical User Interface Screenshot. In Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '18). Association for Computing Machinery, New York, NY, USA, Article 3, 1–6. https://doi.org/10.1145/3220134.3220135

[3] Emil Wallner (2018, Feb 5). How you can train an AI to convert your design mockups into HTML and CSS. https://emilwallner.medium.com/how-you-can-train-an-ai-to-convert-your-design-mockups-into-html-and-css-cc7afd82fed4

[4] Ferreira, J.S. (2019). Live web prototypes from hand-drawn mockups.

[5] Ashwin Kumar, (2018, March 20). SketchCode: Go from idea to HTML in 5 seconds. https://blog.insightdatascience.com/automated-front-end-development-using-deep-learning-3169dd086e82

[6] Noah Gundotra. (2019, April 26). Code2Pix: Deep Learning Compiler for Graphical User Interfaces. Towards Data Science. https://towardsdatascience.com/code2pix-deep-learning-compiler-for-graphical-user-interfaces-1256c346950b

[7] Riaz, Saman & Arshad, Ali & S. Band, Shahab & Mosavi, Amir. (2022). Transforming Hand Drawn Wireframes into Front-End Code with Deep Learning. Computers, Materials & Continua. 72. 4303-4321. 10.32604/cmc.2022.024819.

[8] Robinson, Alex. (2019). Sketch2code: Generating a website from a paper mockup.

[9] Tommaso Calò and Luigi De Russis. 2022. Style-Aware Sketch-to-Code Conversion for the Web. In Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '22 Companion). Association for Computing Machinery, New York, NY, USA, 44–47. https://doi.org/10.1145/3531706.3536462

[10] Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. 3rd International Conference on Learning Representations (ICLR 2015), 1–14.

[11] James a. Landay. Interactive sketching for user interface design. Conference companion on Human factors in computing systems - CHI ’95, pages 63–64, 1995.

[12] Rubine, D. Specifying gestures by example. Computer Graphics 25, 3 (July 1991), 329-337, ACM SIGGRAPH '91 Conference Proceedings.

[13] Saad Hassan, Manan Arya, Ujjwal Bhardwaj, and Silica Kole. Extraction and Classification of User Interface Components from an Image. International Journal of Pure and Applied Mathematics, 118 (24) :1–16, 2018.

[14] Airbnb. (2014, June 12). Building a Visual Language. Airbnb Design. https://airbnb.design/building-a-visual-language/

[15] T. Silva da Silva, A. Martin, F. Maurer and M. Silveira, "User-Centered Design and Agile Methods: A Systematic Review," *2011 Agile Conference*, Salt Lake City, UT, USA, 2011, pp. 77-86, doi: 10.1109/AGILE.2011.24.

[16] Wiriyathammabhum, P., Summers-Stay, D., Fermüller, C., & Aloimonos, Y. (2016). Computer Vision and Natural Language Processing. ACM Computing Surveys (CSUR), 49, 1 - 44.

[17] Jitendra Malik, Pablo Arbeláez, João Carreira, Katerina Fragkiadaki, Ross Girshick, Georgia Gkioxari, Saurabh Gupta, Bharath Hariharan, Abhishek Kar, Shubham Tulsiani, The three R’s of computer vision: Recognition, reconstruction and reorganization, Pattern Recognition Letters

[18] Nadeem (2021, March 10). Encoders-Decoders, Sequence to Sequence Architecture. https://medium.com/analytics-vidhya/encoders-decoders-sequence-to-sequence-architecture-5644efbb3392

[19] fdeloche, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wik

[20] Suresh, K.R., Jarapala, A. & Sudeep, P.V. Image Captioning Encoder–Decoder Models Using CNN-RNN Architectures: A Comparative Study. Circuits Syst Signal Process 41, 5719–5742 (2022). https://doi.org/10.1007/s00034-022-02050-2

[21] Brownlee, J. (2019 April). Caption Generation with Inject and Merge Architectures for Encoder-Decoder Model. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/caption-generation-inject-merge-architectures-encoder-decoder-model/

[22] Li, Y., Ouyang, W., Zhou, B., & Wang, K. (2017). Scene graph generation from objects, phrases and region captions. arXiv preprint arXiv:1708.0

[23] A. Kazlouski and R. K. Sadykhov, "Plain objects detection in image based on a contour tracing algorithm in a binary image," *2014 IEEE International Symposium on Innovations in Intelligent Systems and Applications (INISTA) Proceedings*, Alberobello, Italy, 2014, pp. 242-248, doi: 10.1109/INISTA.2014.6873624.

[24] Z. Zong and C. Hong, "On Application of Natural Language Processing in Machine Translation," *2018 3rd International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, Huhhot, China, 2018, pp. 506-510, doi: 10.1109/ICMCCE.2018.00112.

[25] Wiriyathammabhum, Peratham & Summers Stay, Douglas & Fermüller, Cornelia & Aloimonos, Yiannis. (2016). Computer Vision and Natural Language Processing: Recent Approaches in Multimedia and Robotics. ACM Computing Surveys. 49. 1-44. 10.1145/3009906.

[26] K. Gouhara, T. Watanabe and Y. Uchikawa, "Learning process of recurrent neural networks," *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, Singapore, 1991, pp. 746-751 vol.1, doi: 10.1109/IJCNN.1991.170489.

[27] Z. Quan, W. Zeng, X. Li, Y. Liu, Y. Yu and W. Yang, "Recurrent Neural Networks With External Addressable Long-Term and Working Memory for Learning Long-Term Dependences," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 3, pp. 813-826, March 2020, doi: 10.1109/TNNLS.2019.2910302.

[28] Shipra Saxena. (2021, March 16). Learn about Long Short-Term Memory (LSTM) Algorithms. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

[29] R. Huang and X. Ren, "Research on the Application of Time Series Forecasting Model Based on Encoder-Decoder LSTM Model," *2022 IEEE 4th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, Dali, China, 2022, pp. 221-224, doi: 10.1109/ICCASIT55263.2022.9986527.

[30] Gackenheimer, C. (2015). Introduction to React. *Apress*.

[31] Markus Freitag, Yaser Al-Onaizan . (2017, June 14). Beam Search Strategies for Neural Machine Translation. arXiv preprint arXiv:1702.01806.

[32] Haibing Wu, Xiaodong Gu. (2015, Dec 4). Max -Pooling Dropout for Regularization of Convolutional Neural Networks. arXiv:1512.01400 [cs.LG