TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

Major Project Report On
User Interface Code Generation from Hand-drawn Sketch

Submitted By:

Manoj Paudel	THA077BCT025
Prince Poudel	THA077BCT036
Ronish Shrestha	THA077BCT040
Sonish Poudel	THA077BCT042

Submitted To:
Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

March 2025

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

Major Project Report On
User Interface Code Generation from Hand-drawn Sketch

Submitted By:

Manoj Paudel	THA077BCT025
Prince Poudel	THA077BCT036
Ronish Shrestha	THA077BCT040
Sonish Poudel	THA077BCT042

Submitted To:
Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor's Degree in Computer Engineering.

Under the Supervision of
Er. Devendra Kathayat

March 2025

DECLARATION

We hereby declare that the report of the project entitled **"User Interface Code Generation from Hand-drawn Sketch"** which is being submitted to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, in the partial fulfillment of the requirements for the award of the Degree of Bachelor of Engineering in **Computer**, is a bonafide report of the work carried out by us. The materials contained in this report have not been submitted to any University or Institution for the award of any degree and we are the only author of this complete work and no sources other than the listed here have been used in this work.

Manoj Paudel (Class Roll No:THA077BCT025) _____

Prince Poudel (Class Roll No:THA077BCT036) _____

Ronish Shrestha (Class Roll No:THA077BCT040) _____

Sonish Poudel (Class Roll No:THA077BCT042) _____

Date: March 2025

CERTIFICATE OF APPROVAL

The undersigned certify that they have read and recommended to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, a major project work titled **"User Interface Code Generation from Hand-drawn Sketch"** submitted by **Manoj Paudel, Prince Poudel, Ronish Shrestha, Sonish Poudel** in partial fulfillment for the award of Bachelor's Degree in Computer Enginnering. The Project was carried out under special supervision and within the time frame prescribed by the syllabus.

We found the students to be hardworking, skilled and ready to undertake any related work to their field of study and hence we recommend the award of partial fulfillment of Bachelor's Degree in Computer Enginnering.

Project Supervisor
Er. Devendra Kathayat
Department of Electronics and Computer Engineering, Thapathali Campus

---

External Examiner
Dr. Prakash Poudyal
Department of Computer Science and Engineering, Kathmandu University

---

Project Co-ordinator
Er. Saroj Shakya
Department of Electronics and Computer Engineering, Thapathali Campus

---

Head of the Department
Er. Umesh Kanta Ghimire
Department of Electronics and Computer Engineering, Thapathali Campus

March 2025

COPYRIGHT

ACKNOWLEDGMENT

We would like to express our sincere gratitude towards the Institute of Engineering, Tribhuvan University for the inclusion of major project in the course of Bachelors in Computer Engineering. We are also thankful towards our Department of Electronics and Computer Engineering for the proper orientation and guidance during the project **"User Interface Code Generation from Hand-drawn Sketch"**.

We would like to acknowledge the authors of various research papers and developers of various programming libraries and frameworks that we have reference for developing our project. We would like to express our gratitude towards our Project Supervisor **Er. Devendra Kathayat** for continuous suggestions throughout. Finally, we would like to thank all the people who are directly or indirectly related during our study and preparation of this project.

Sincerely,

Manoj Paudel (Class Roll No:THA077BCT025)

Prince Poudel (Class Roll No:THA077BCT036)

Ronish Shrestha (Class Roll No:THA077BCT040)

Sonish Poudel (Class Roll No:THA077BCT042)

ABSTRACT

This report presents "User Interface Code Generation from Hand-drawn Sketch," where we developed an artificial intelligence model that generates HTML/CSS code from layout sketches. A transformer model was implemented where the input is a sketch and the output is DSL (Domain Specific Language) code. A custom DSL was developed consisting of elements such as header, image, and text that describe sketches with hierarchical information. As transformer model requires large datasets, a dataset generator was created that produces sketches based on given DSL code, with DSL code created using various element combinations. The model follows an encoder-decoder architecture where the encoder processes the input image and the decoder transforms it into DSL code. Three models with different encoders were trained and evaluated: Compact Convolutional Transformer Encoder, Vision Transformer, and Convolutional Encoder. Among these, the Compact Convolutional Transformer Encoder performed best with a BLEU score of 0.901 in optimal conditions. A Graphical User Interface was also developed allowing users to customize text, images, and fonts according to their preferences, with functionality to download the resulting HTML/CSS files. Future work could enhance this project through JavaScript integration to add more features.

Keywords: Artificial Intelligence, Domain Specific Language (DSL), Image Processing, Image synthesis, Self-Attention, Transformer Decoder, Vision Transformer (ViT)

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BLEU	Bilingual Evaluation Understudy
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DSL	Domain Specific Language
DSLR	Digital Single Lens-Reflex
FFNN	Feed-Forward Neural Network
GPU	Graphic Processing Unit
GUI	Graphical User Interface
LDPs	Local Directional Patterns
LN	Layer Normalization
ML	Machine Learning
MP	Mega Pixel
NLP	Natural Language Processing
NumPy	Numerical Python
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
UI	User Interface
UX	User Experience
ViT	Vision Transformer

1 INTRODUCTION

1.1 Background

Traditionally, in the field of software development, the creation of user interfaces (User Interface (UI)) has relied heavily on manual processes. Developers have been tasked with interpreting design specifications and translating them into functional code. This method often involves manually coding UI elements such as buttons, input fields, menus, and layouts. While this approach has been effective in bringing designs to life, it is both time-consuming and prone to introducing errors. The discrepancies between the designer's vision and the developer's implementation can result in interfaces that deviate significantly from the original design. Recognizing these limitations, the "Sketch-to-Code" system proposes a new approach that automates the conversion of design sketches into executable front-end code, thereby addressing these persistent challenges. The reliance on manual coding for UI development has long been a bottleneck in the software development process. Designers often go to great lengths to craft meticulously detailed and visually consistent design systems. However, these designs often fail to seamlessly translate into the development phase. The variations introduced during manual implementation not only waste valuable time but also lead to inconsistencies that can compromise the overall user experience. The "Sketch-to-Code" system aims to bridge this gap by providing a tool that leverages advanced technologies to automate the transformation of design sketches into functional code. This solution not only enhances efficiency but also ensures that the implemented interface remains faithful to the original design.

Recent advancements in technology, particularly in the fields of Artificial Intelligence (AI) and machine learning, have significantly impacted the way Graphical User Interface (GUI) sketches are converted into functional code. Researchers and developers are increasingly exploring the potential of these technologies to streamline UI development. By utilizing sophisticated algorithms and image processing techniques, automated systems are now capable of analyzing design sketches, interpreting their components, and generating corresponding code. These systems offer several key advantages. First, they drastically reduce the time required for UI development, enabling developers to focus on more complex and creative tasks. Second, they ensure a higher level of consistency between the design and the implemented interface, thereby reducing the need for extensive revisions. Finally, they allow designers and developers to collaborate more effectively, as the automated system serves as a bridge that translates design specifications into code with minimal manual intervention.One of the most notable benefits of the "Sketch-to-Code" system is its ability to enhance the overall efficiency of the

software development process. By automating repetitive and time-consuming tasks, developers can allocate their efforts toward refining the functionality and user experience of the application. This shift in focus not only accelerates project timelines but also results in higher-quality software products. Furthermore, the consistency ensured by the system eliminates many of the errors that typically arise during manual implementation, thereby reducing the need for extensive testing and debugging.The integration of AI into UI development marks a significant milestone in the evolution of software engineering practices. As these AI-driven solutions continue to advance, they hold the promise of revolutionizing the way software is designed and developed. By making UI development more accessible and efficient, these technologies have the potential to empower developers across a wide range of industries. Whether it is a small startup looking to prototype a new idea or a large enterprise aiming to streamline its development pipeline, the ”Sketch-to-Code” system offers a powerful tool to achieve these goals.In conclusion, the ”Sketch-to-Code” system represents a transformative approach to UI development. By automating the conversion of design sketches into functional code, it addresses many of the challenges that have long plagued designers and developers. With the support of AI and machine learning, this system not only enhances efficiency and consistency but also paves the way for more innovative and user-centric software solutions. As the technology continues to evolve, it is poised to become an integral part of modern software development, enabling teams to bring their creative visions to life with greater speed and precision.

1.2 Motivation

This project is focused on addressing a common and time-consuming issue in the software development process: transforming design ideas into fully functional software. Often, developers encounter significant challenges when the final product fails to align with the original design vision. These mismatches typically result in repeated adjustments and rework, which not only disrupts the development timeline but also leads to inefficiencies and delays in project completion. To tackle this issue, our approach involves providing a simple and effective solution in the form of prototype front-end code. By automating the initial steps of creating user UI, we aim to streamline this process, making it quicker and more efficient for developers to get started on their projects.By leveraging automation, this project seeks to reduce the technical burden associated with manually building UI components such as buttons, menus, and layouts. Instead of spending valuable time on these repetitive and detailed tasks, developers will have the opportunity to focus on enhancing the overall functionality and user experience of the software. This shift not only accelerates the development process but also ensures that the final product remains true to the original design vision, minimizing discrepancies

and the need for extensive revisions.

The ultimate goal of this project is to simplify and enhance the software development workflow. By providing developers with tools that bridge the gap between design and implementation, we aim to create a more seamless and productive development experience. This approach not only saves time but also ensures that the end product aligns closely with the intended design, eliminating unnecessary variations and preserving the integrity of the original vision. Through these advancements, this project aspires to empower developers to deliver high-quality software more efficiently and effectively.

1.3 Problem Definition

The problem is to develop a system that can automate the process of turning sketches into functional front-end code. The system should be able to craft interactive UI that is more like designer intended. Some of the key challenges are listed below:

- **Ambiguity in Requirements:** Sketches may not always clearly define all requirements, leading to ambiguity. This can result in misunderstandings between stakeholders, designers, and developers.
- **Technical Feasibility:** Sometimes, what looks good on paper (or screen) may not be technically feasible within the constraints of the project, platform, or technology stack.
- **Integration Complexity:** Projects often require integrating various components, APIs, or third-party services, which can introduce complexity beyond the initial sketch.
- **User Experience (UX) Considerations:** Implementing a sketch into code involves not just visual fidelity but also ensuring a smooth and intuitive user experience, which may require iterative improvements.

1.4 Objectives

The main objectives of our project are listed below :

- To construct a model able to generate quick GUI prototype
- To make intutive user interface for customizing and stylizing the generated code.

1.5 Project Scope and Applications

In the world of computer science, most projects start with a concept or idea. These ideas are often visualized through sketches, diagrams, or mockups. These sketches serve as an important part of the project's foundation, helping to define the project's main goals, user interfaces, and the architecture of the system. They represent a high-level overview of how the final product should look and function. The main challenge is to take these static sketches and transform them into working, functional code that fulfills the intended goals of the project, aligns with the requirements, and meets user expectations.

The scope of converting these sketches into code is a complex process. It involves more than just turning images into text-based code. The task requires ensuring that the design is not only visually accurate but also functionally correct. Developers need to ensure that the features and components shown in the sketch are possible to build with the given technology stack, and that they meet the performance and usability expectations. This process covers various areas, including the visual design, user interactions, back-end architecture, and integration of multiple systems or services. The ultimate goal is to create software that functions just as the designer intended, while also being practical, efficient, and responsive.The application of turning sketches into code has several key benefits for the development process. First, it provides a clear, structured path for developers to follow, making sure they know exactly what the final product should look like and how it should work. It allows them to better understand both the functional and aesthetic requirements of the project, which are critical for creating a seamless user experience. By following the sketches, developers can create software that is both visually appealing and user-friendly.When developers adhere to the sketches during the development process, they can ensure that the design remains consistent. Consistency is important because it maintains the intended look and feel of the application across all parts of the project. This consistency reduces the chances of errors or discrepancies arising between different sections of the software, ensuring that all elements work together harmoniously. By starting with clear sketches, the team can avoid confusion and misalignment during the development process, making it easier to make design decisions and solve problems quickly.

In addition to providing direction for developers, sketches also act as a useful communication tool. They provide a visual representation that helps different stakeholders, such as designers, developers, and project managers, understand the project's objectives. This shared understanding is crucial for minimizing misunderstandings and ensuring that everyone is on the same page. It fosters collaboration between teams, allowing

designers to communicate their ideas clearly, developers to ask questions about technical feasibility, and stakeholders to provide input on the project's progress. Having sketches as a reference point helps keep everyone aligned throughout the lifecycle of the project.On a practical level, converting sketches into code involves selecting the best programming languages, frameworks, and tools that suit the project's needs. Developers take the visual elements and translate them into code, ensuring that the application works as expected and performs well. This involves not just writing the code for visual elements but also integrating them with other parts of the system, like databases or APIs. The process requires developers to pay attention to both the technical details and the user experience. They must ensure that the final product not only meets the technical specifications but also provides an interface that is easy to use and navigate.

Another important aspect of the sketch-to-code process is the flexibility it offers in development. By working from the sketches, developers can follow an iterative development approach. This means that the project can evolve over time, based on feedback from users or stakeholders. As feedback is gathered, developers can make improvements to the software, refining it until it meets the needs of the users and aligns with the vision of the project. This iterative approach allows for continuous improvements, making the development process more adaptive and responsive to changing requirements or new insights. Ultimately, the process of translating sketches into code plays a crucial role in ensuring the success of a computer science project. It helps developers create software that is functional, meets the requirements, and provides a positive user experience. By starting with a clear, visual design, the project is guided in a consistent and structured way, reducing the chances of miscommunication or errors. Developers are able to work more efficiently, as they have a solid foundation to build on. And because sketches facilitate collaboration and communication, all stakeholders can be involved in the development process, ensuring that the final product is exactly what the user needs.

2 LITERATURE REVIEW

2.1 Different methods used for this kind of task

The literature review for this research was conducted by searching through various academic repositories such as IEEE Xplore, Google Scholar, and other trusted sources. The search included a variety of keywords such as ”sketch to code,” ”image to code,” ”GUI to functional code,” and others, all relating to the process of converting visual design representations into executable software code. The review primarily focused on research papers, conference proceedings, and relevant articles that detail the methodologies, tools, and technologies used in this field. Along with the academic papers, various existing systems that are attempting to automate this process were also reviewed to understand the evolution and challenges faced in the area of sketch-to-code conversion.

One of the most notable contributions in this area comes from the 2017 paper [1] which introduced Pix2code. Pix2code is a pioneering deep learning-based system designed to convert graphical user interfaces (GUIs) into executable code. By analyzing screenshots of a GUI, Pix2code identifies various components such as buttons, menus, and images, and generates corresponding code in different programming languages. This system bridges the gap between design and development, making it easier for designers to translate their visual ideas into working software components. The key strength of Pix2code lies in its ability to automate code generation, saving valuable time during the prototyping phase of development. However, the system faces challenges such as the need for large, high-quality datasets for training and issues in handling highly complex or non-standard designs. Despite these hurdles, Pix2code remains a significant step forward, making the design-to- development process faster and more efficient by reducing manual coding efforts. The use of deep learning models in Pix2code has been recognized for transforming how the design-to-development workflow is approached.

Moving forward, in 2019, the paper [2] introduced a new approach to transforming hand-drawn sketches into interactive software prototypes. Unlike traditional methods of prototyping, which rely on manual coding or graphic design tools, Eve aimed to automate much of the process by utilizing Machine Learning (ML) and image recognition algorithms. Eve’s approach involves several stages, including User Interface Sketch Recognition, Prototype Generation, and User Interaction, all of which contribute to generating a functional prototype from a simple sketch. By leveraging ML algorithms, the system recognizes UI components in hand-drawn sketches and transforms them into working software prototypes. This system provides a faster alternative to traditional prototyping methods, significantly reducing development time. However, while Eve presents an innovative solution, the paper highlights challenges such as achiev-

ing recognition accuracy, handling the complexity of real-world UI designs, and ensuring the system's scalability and usability in real-world applications. Despite these challenges, the paper presents a promising methodology for automating the sketch-to-prototype process, which can greatly benefit the software development community by enabling faster iterations and reducing dependency on expert coding skills.

In 2020, another groundbreaking paper [3]took a step further in automating the sketch-to-code process by focusing on low-fidelity, hand-drawn sketches of GUIs. The system utilizes advanced deep learning models, particularly convolutional neural networks (CNNs) for image recognition, and Long Short-Term Memory (LSTM) networks for sequence prediction. The goal of this research was to improve the accuracy and efficiency of converting simple sketches into functional code. CNNs are used to recognize and identify different UI elements, while LSTMs help generate the corresponding code sequences needed for the final output. By combining these two techniques, the system is better equipped to handle complex UI layouts and intricate designs, which earlier models struggled to interpret correctly. This model also aims to handle dense and detailed sketches, where the layout and UI components may overlap or be difficult to distinguish. However, while the approach provides significant improvements, the system still faces issues in interpreting diverse sketch styles and integrating the generated code into existing development workflows. These challenges point to the complexity of designing a system that can accurately process a wide range of sketch styles while producing functional, high-quality code. Despite the challenges, the research represents an important step in improving the process of generating code from graphical user interfaces and pushing the boundaries of sketch-to-code systems.

Building on these advancements in 2021, the paper [4] introduced a system focused on mobile app development. The aim was to allow users to sketch mobile app interfaces, which would then be automatically converted into functional code. The research aimed to democratize the development process, enabling users without programming knowledge to create mobile applications. This system uses CNNs for accurate recognition of various UI elements and LSTMs to generate the code that corresponds to the design. The end goal of this research is to empower end-users by providing them with tools to create applications without needing to learn complex coding languages. However, similar to earlier models, this system faces challenges, such as accurately recognizing different sketch styles, ensuring the generated code functions as expected, and integrating it seamlessly into mobile app development environments. While the research marks an important step in enabling non-technical users to participate in mobile app creation, the system's challenges highlight the complexities of converting hand-drawn sketches into functional code that can be used in real-world development. Nonetheless,

it represents a significant breakthrough in user-friendly development tools for mobile apps.

Similarly in 2022, the paper [5] made a significant contribution to the field of automated code generation. This research proposed a multi-step framework designed to generate skeleton code for full-stack applications from hand-drawn sketches. The framework incorporated advanced deep learning techniques, including convolutional neural networks (Convolutional Neural Network (CNN)), to identify and classify user interface (UI) components such as buttons, text fields, and menus. It also leveraged advanced image processing algorithms to enhance the accuracy of element recognition. Once the UI elements were identified, the system mapped these components to corresponding front-end and back-end code modules.The innovation of Sketch2FullStack lay in its ability to streamline the development process by bridging the gap between conceptual design and implementation. The generated skeleton code provided developers with a strong foundation, reducing the time spent on initial setup and allowing for greater focus on customization and functionality. The research addressed challenges such as handling complex layouts and integrating multiple layers of code across the stack, from front-end HTML and CSS to back-end APIs and databases. However, the system encountered limitations in interpreting sketches with intricate designs or unconventional layouts, highlighting the need for further advancements in deep learning algorithms. This work demonstrated how automated tools could significantly improve developer productivity while maintaining alignment with design specifications.

Moreover in 2023, the paper [6] introduced a novel methodology for transforming sketches into executable code. Unlike prior systems, SkCoder emulated the behavior of developers by utilizing an intelligent code-retrieval approach. The system was designed to search for similar code snippets in large-scale repositories, extract relevant parts, and adapt them to match the structure and style of the sketch provided by the user. SkCoder combined machine learning models with heuristic-based rules to refine the retrieved code and ensure its functionality within the project's context.A key feature of SkCoder was its ability to incorporate user feedback iteratively, allowing developers to fine-tune the generated code. This feedback loop enabled the system to learn from corrections, progressively improving its accuracy and relevance. The research highlighted the importance of balancing automated generation with developer intervention to address edge cases and maintain code quality.Additionally, SkCoder placed a strong emphasis on enhancing collaboration between designers and developers. By providing a semi-automated solution, the tool enabled designers to visualize how their sketches would translate into actual code while offering developers a solid starting point for implementation. The system demonstrated particular effectiveness in handling standard

UI components and layouts but faced challenges when dealing with non-standardized or overly creative designs. Nonetheless, SkCoder represented a significant step forward in creating intelligent and adaptable tools for modern software development.

More recently, in 2024, a paper [7] was published, offering another perspective on the sketch-to-code problem by focusing on the generation of HTML and CSS code from hand-drawn sketches. This paper addresses the need for automation in web development by proposing a system that uses deep learning techniques to convert hand-drawn UI sketches into working web pages. The system goes through a four-phase process: pre-processing, segmentation, feature extraction, and classification, with each phase designed to interpret the visual features of the sketch and categorize them into HTML and CSS elements. Local Directional Patterns (LDPs) are used in the feature extraction phase to capture detailed visual patterns, enhancing the system's ability to correctly classify UI components. While earlier methods in this domain were based on rule-based systems, which often lacked flexibility, this new system introduces a more robust and adaptable approach. However, like its predecessors, it faces challenges related to scalability, accuracy, and handling diverse sketch styles. Despite these challenges, the paper provides an important contribution to the ongoing development of systems capable of automatically converting sketches into functional web code.

2.2 Vision Transformer for Computer vision

In addition to the above works, a significant contribution to the field of computer vision was made by Dosovitskiy et al. in 2021 with their paper [8] This research introduced the ViT, a novel deep learning model designed to process images in a way similar to how Natural Language Processing (NLP) models handle text. Unlike traditional CNN, which use filters to process images, the ViT model treats images as a sequence of patches, each of which is classified using transformer models. The ViT model has achieved outstanding results in large-scale image classification tasks, and its principles have been adapted for use in sketch-to-code systems. By using transformers, researchers have been able to improve the accuracy of image recognition tasks and enhance the ability of sketch-to-code systems to interpret complex visual data. While the Vision Transformer model holds great potential for improving sketch-to-code systems, challenges still remain in fine-tuning the model to process UI designs accurately and ensuring it integrates seamlessly with the code generation process.

In conclusion, the literature reveals substantial progress in the development of systems that automate the translation of sketches into executable code. From the early rule-based methods to more recent deep learning-based approaches, significant strides have been

made in enhancing the accuracy, scalability, and efficiency of these systems. Despite the advancements, challenges persist, particularly in handling diverse sketch styles, improving recognition accuracy, and ensuring seamless integration with real-world development workflows. Nonetheless, the research in this domain continues to evolve, with new methodologies and models.

3 REQUIREMENT ANALYSIS

3.1 Software Requirement

In order to implement our project of automating the process of converting sketches into functional front-end code, the following software tools and libraries are indispensable. Each of these tools has been chosen for its critical role in different stages of the project:

3.1.1 Python

Python is a high-level, versatile programming language that has become a cornerstone in fields such as artificial intelligence, machine learning, data science, and software development. Its design philosophy emphasizes readability and simplicity, enabling developers to focus on solving problems rather than managing code complexity. Python boasts an extensive ecosystem of libraries and frameworks that make it highly adaptable for implementing cutting-edge solutions.In this project, Python serves as the foundation for every aspect of development, from preprocessing sketch data to training machine learning models and generating functional code. Its cross-platform compatibility allows smooth integration with various tools and frameworks. Furthermore, Python's active open-source community ensures regular updates, abundant resources, and support, making it ideal for building a robust and scalable solution.

3.1.2 Numpy

Numerical Python (NumPy) is a core library for numerical and scientific computing in Python. It is designed to provide efficient support for handling large, multi-dimensional arrays and matrices, which are essential for processing image data and performing computations in machine learning workflows. NumPy's optimized performance ensures that computational tasks involving large datasets can be executed quickly and reliably.In our project, NumPy plays a significant role in processing images of hand-drawn sketches. It enables efficient manipulation of pixel data, normalization of input images, and transformations such as resizing and cropping. Additionally, NumPy is a critical component in data preparation, where it facilitates mathematical operations that prepare data for feeding into deep learning models. Its seamless integration with libraries like TensorFlow further simplifies the workflow and ensures consistency.

3.1.3 Tensorflow

TensorFlow is an open-source machine learning framework developed by Google. It is one of the most widely used platforms for building, training, and deploying machine learning and deep learning models. TensorFlow provides powerful tools for managing tensors, which are multi-dimensional arrays that serve as the backbone of neural

network operations. It supports automatic differentiation, optimization, and distributed training, making it a versatile choice for handling large-scale projects.For this project, TensorFlow acts as the core engine for implementing the deep learning models that drive the conversion of sketches into functional code. It allows us to design sophisticated convolutional neural networks (CNN) to identify UI elements in sketches, and sequence models like Recurrent Neural Network (RNN) or transformers for generating structured code. TensorFlow's scalability enables us to train models on high-performance hardware such as GPUs, ensuring the project can handle the computational demands of large datasets and complex designs. Additionally, TensorFlow's deployment tools make it easier to integrate the trained models into production environments.

3.1.4 Keras

Keras is a high-level neural network Application Programming Interface (API) built on top of TensorFlow, offering a simplified and user-friendly interface for creating and training deep learning models. Keras provides pre-built components such as layers, optimizers, and loss functions, making it accessible even for developers who are not experts in machine learning. Its modular structure allows users to experiment with different architectures and hyperparameters with minimal effort.In the context of this project, Keras is essential for building and fine-tuning neural networks designed to interpret sketch data and generate front-end code. It allows for rapid prototyping and testing of models, enabling developers to iterate quickly and find optimal solutions. Keras also includes tools for visualizing model performance, such as training accuracy and loss, which are crucial for diagnosing issues and improving model reliability. By leveraging Keras, we can streamline the development of deep learning pipelines and maintain focus on achieving project goals without being bogged down by implementation details.

3.2 Hardware Requirements

The hardware requirements for this project are essential to ensure the effective collection of input data and the efficient training of deep learning models. Two primary hardware components are required: a camera device for capturing sketched designs and a cloud computing platform for performing computationally intensive training tasks. Together, these components form the backbone of the system's functionality and performance.

3.2.1 Camera Device

For the successful digitization of manually sketched designs, a high-quality camera device is required. The minimum recommended resolution for the camera is 12 Mega Pixel (MP), which is commonly found in most modern smartphones. Such devices are capable of producing clear and detailed images that are sufficient for analysis and feature extraction. However, using a camera with a higher resolution, such as 48 MP or 108 MP, can further enhance the quality of the images, capturing fine details of intricate sketches that may otherwise be lost. In professional settings, Digital Single Lens-Reflex (DSLR) or mirrorless cameras can be employed to achieve even better results. These advanced cameras offer superior image quality and features like manual focus, adjustable exposure, and higher dynamic range. Additionally, features such as image stabilization and customizable settings for white balance and ISO ensure distortion-free and sharp images, even under challenging lighting conditions. To maximize accuracy, it is recommended to capture images in a well-lit environment, as proper lighting significantly improves the visibility and clarity of sketch details.

3.2.2 Cloud Computing Platform

The training of deep learning models for this project requires computational resources beyond what standard personal computers can provide. A robust cloud computing platform with access to dedicated Graphic Processing Unit (GPU) and Central Processing Unit (CPU) resources is essential for handling large datasets and performing complex operations. The recommended platforms include Google Colab and Kaggle, which provide free hosted environments equipped with powerful GPU resources, such as NVIDIA Tesla T4 or K80. These platforms are well-suited for deep learning tasks and allow developers to focus on model building and experimentation without worrying about hardware constraints. Google Colab offers seamless integration with Google Drive, allowing easy storage and retrieval of datasets and model checkpoints. Its support for collaboration makes it an ideal choice for teams working on the project. Similarly, Kaggle provides access to community-shared datasets and pre-trained models, which

can accelerate development and reduce the time required for data preparation.

For larger-scale projects or cases where free resources are insufficient, paid platforms like AWS, Google Cloud Platform, or Microsoft Azure can be utilized. These platforms offer flexible configurations, including high-performance GPU instances like NVIDIA V100 or A100, enabling faster and more efficient model training. Additionally, they provide advanced features such as automated scaling and integration with machine learning pipelines, ensuring that the system can adapt to evolving requirements during development. By leveraging these hardware components effectively, the project can achieve high accuracy in sketch analysis and efficient training, ensuring the final system meets both technical standards and user expectations.

4 SYSTEM ARCHITECTURE AND METHODOLOGY

4.1 Dataset

For the successful implementation of the approach, a carefully maintained dataset is required, consisting of wireframe sketches paired with their corresponding DSL code. These sketches and codes are fundamental for training the machine learning model to recognize and translate visual designs into functional code. However, a pre-existing dataset containing both wireframe sketches and the associated DSL code could not be found. It was determined during the research that no publicly available dataset fully met the specific requirements of the project. As a result, the decision was made to create a custom dataset from scratch. This ensured that the dataset closely aligns with the goals of the project and contains the necessary variety and diversity for effective model training.

4.1.1 Dataset Generator

The dataset generator plays a central role in the project, as it is responsible for producing a vast and varied set of training samples that will be used by the model. This generator operates through a structured, multi-step process designed to create the necessary wireframe sketches and their corresponding DSL code. The process begins with the random generation of DSL code, which specifies the structural elements and layout of a web page. The generated DSL code is then compiled into HTML and CSS, which are used to render a visual representation of the web page layout. Once the visual layout is created, the rendered page is analyzed to identify and map out the positions and boundaries of each individual element. These identified positions are crucial for accurately placing the corresponding visual elements in the next phase. In the final step of the process, hand-drawn sketches are placed onto the generated layout at the appropriate positions. These sketches mimic the real-world designs of web pages, but in a simplified, hand-drawn style, closely resembling wireframe designs.This automated process allows for the production of a large number of unique and diverse sketch-DSL pairs. Each pair is representative of realistic, real-world web page designs, and the generator ensures that these pairs are varied in structure, complexity, and design style. By utilizing predefined rules and constraints, the dataset generator ensures that the generated layouts are not only realistic but also diverse, capturing the wide range of design possibilities that exist in the real world. With this automated dataset generation process, as many samples as required for training the model can be efficiently generated without relying on manual data collection. This ensures that the model is exposed to a broad set of design styles and can accurately learn to translate wireframe sketches into functional DSL code.

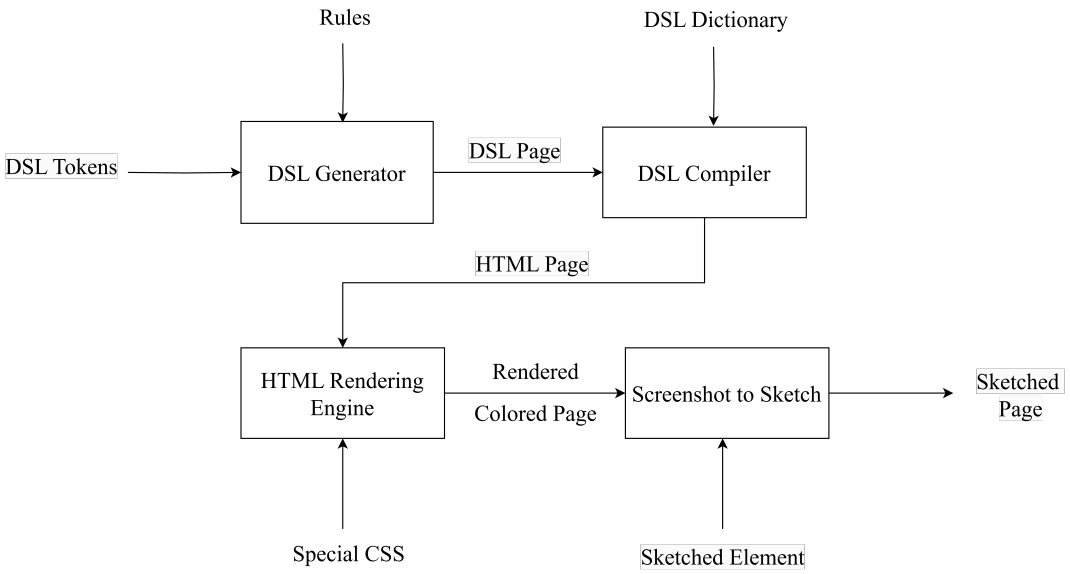

Figure 4-1: Block Diagram of Data Synthesis

4.2 Block Diagram

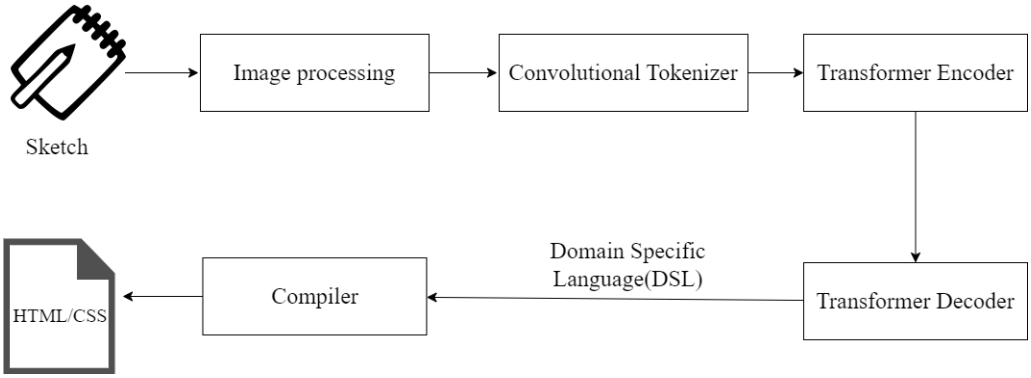

Figure 4-2: Block Diagram of Working of System

4.3 Working Principle

The core principle behind this project is to transform a manually created wireframe sketch of a user interface (UI) into functional HTML code. The process involves several key stages, each aimed at capturing the essential visual components of the sketch and translating them into corresponding HTML elements. The following steps outline the approach taken to achieve this transformation:

i. **DSL Code Extraction using Transformer Model:**

The first step in the process involves extracting the domain-specific language (DSL) code from the input sketch. The sketch, which serves as a visual representation of the UI, is analyzed using a deep learning model, specifically a Trans-

former model, which has been trained to recognize the structure and components of the design. The Transformer model is particularly useful in understanding the spatial relationships between different UI elements in the sketch. Through a process of visual recognition and feature extraction, the model identifies components such as buttons, text fields, images, and containers, and generates the corresponding DSL code. This code is a simplified representation of the layout, defining the structure and properties of each UI element.

ii. **Compiling DSL Code to HTML Code:** Once the DSL code has been extracted, the next step is to compile it into HTML code. The generated DSL code serves as a blueprint for the layout, and each DSL element corresponds to a specific HTML tag or structure. This step involves mapping the DSL elements to their respective HTML equivalents, ensuring that the visual design described by the DSL is accurately translated into a functional HTML layout. This step may involve additional styling elements, such as CSS for positioning and design tweaks, to ensure the generated HTML code matches the intended appearance of the original sketch. The process in this project has two main steps—first, extracting the DSL code using a Transformer model, and then converting it into HTML code. This is the main idea behind how the system works. It automates the process of turning a simple sketch into functional front-end code, making it much easier for developers to create prototypes quickly.Using ML techniques like Transformer models, the system can recognize and understand the layout of the design. After that, it converts it into HTML, making sure the design works well in a web browser. This method speeds up development and helps designers and developers communicate better, as it turns a design concept directly into working code.

4.3.1 Image Processing

Preprocessing is required to translate an image from a camera into an image which can be fed into the model. Due to the positioning of the camera or lighting conditions the raw image must be cleaned up before it can be processed.

The main challenges are:

- The image may not fill the entire frame, as such the background must be removed.
- The paper may be skewed or rotated.
- The image may contain noise or alterations due to lighting.

For solving above challenges:

- The paper was detected in the image and cropped. As the requirements stated that the medium must be white, the image was converted to HSV, and threshold filtering was applied to remove all colors except white. This process considerably reduced the background noise. Since the paper contained the sketch in a dark marker, these pixels were filtered out using threshold filtering. To fill the gaps left by the sketch, a large median blur was applied. The edge map was then dilated to close small gaps between the edges. Finally, contour detection was applied, and the largest contour with approximately four sides was identified, as it was assumed the medium was four-sided (e.g., paper or a whiteboard).

- Perspective warping was applied to unwrap the four corners of the contour found and map them to an equivalent rectangle, thereby correcting the orientation of the page.

- A median blur was applied to the unwrapped cropped image to reduce noise. The edge map was then dilated to close small gaps between lines. The result of the post-processing was an unskewed binary edge map of the sketch, which was subsequently fed into the model for processing.

4.3.2 Convolutional Tokenizer

A convolutional tokenizer is an important component in neural networks, particularly when working with images. This part of the system processes visual data, applying convolutional layers to detect and capture local patterns and features within an image. Convolutional layers work by scanning the image with small filters, which help identify important features such as edges, textures, or shapes at different scales. After applying these filters, the network often uses pooling layers, which reduce the size of the image while keeping the most relevant information intact. This helps make the process more efficient by focusing on the key aspects of the image.The outcome of these convolutional operations is a set of feature maps, which are essentially condensed representations of the image. These feature maps contain high-level abstractions of what is found in the image, making them easier to analyze and interpret. Once the feature maps are generated, they are reshaped or flattened into a sequence of vectors, which are called tokens. Each token represents a specific part of the image, such as a button, a menu, or a text box in the context of UI design. This sequence of tokens can now be treated as input for sequence models, similar to how natural language processing models work with text. By doing so, the convolutional tokenizer enables the application of advanced techniques, like transformers, to image data, which would otherwise be difficult to process.

4.3.3 Transformer Model

The Transformer model is a powerful deep learning architecture widely used for tasks involving sequences of data. Its encoder-decoder structure makes it particularly effective for these tasks. The encoder processes the input sequence, with each token representing a feature of the image, and converts them into continuous representations that capture the essential information. Once the encoder has processed the tokens, the decoder generates an output sequence, which, in this case, is the Domain-Specific Language (DSL) code. The decoder uses the encoded information to produce a sequence of symbols that describe the structure of the image or user interface.The Transformer model is designed with self-attention mechanisms, enabling it to focus on different parts of the input sequence simultaneously. This makes the model efficient at understanding complex relationships in data. The self-attention layers, along with fully connected layers, allow the model to capture the most relevant information from the input, even when the data is spread out or has long-range dependencies.In tasks like image captioning, the input is an image, and the output is a text description of what the image contains. However, in this case, the output is a structural description of the user interface in DSL, which defines the layout and behavior of different UI elements. This allows for the generation of detailed, accurate code that represents the structure of the UI, making it easier to convert sketches into functional front-end code.

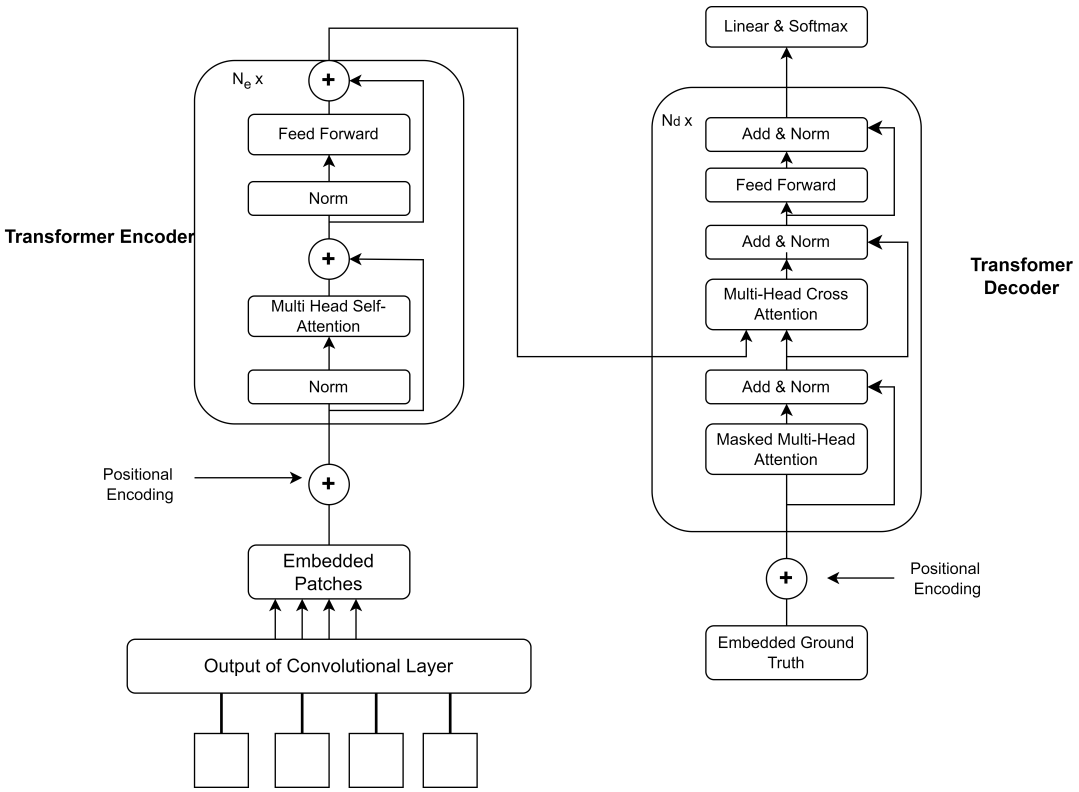

Figure 4-3: Block Diagram of Transformer Model

Transformer Encoder

The Transformer encoder is a fundamental part of the Transformer architecture, originally developed for NLP tasks but has since been adapted and widely used in various domains, including Computer Vision (CV). The encoder consists of multiple identical layers, with each layer containing two critical subcomponents: the multi-head self-attention mechanism and the position-wise fully connected feed-forward network. These two subcomponents are designed to handle different aspects of processing the input sequence and help the model learn complex patterns in the data. The multi-head self-attention mechanism is essential because it enables the model to evaluate the relationships between different elements in the input sequence, regardless of their positions or distance from one another. This is a significant advantage over previous architectures like RNN, which processed data sequentially and struggled with long-range dependencies. By using multiple attention heads, the Transformer can simultaneously focus on different parts of the input sequence, capturing diverse relationships and dependencies at different levels of abstraction. This makes it particularly effective for tasks where understanding the context between distant elements in the data is important, such as understanding the layout and structure of an image in computer vision tasks. The position-wise feed-forward network, which follows the self-attention layer, processes each element independently, applying non-linear transformations to increase the model's capacity to learn more complex functions. This additional step allows the model to better understand intricate patterns within the data. Each position in the sequence is transformed individually, allowing the network to adapt to the unique features of each token. The encoder's architecture also employs layer normalization and residual connections around each sub-layer, which greatly assist in stabilizing and accelerating the training of deep networks. These techniques make it easier for the model to converge and learn effectively, especially when working with large datasets and complex tasks like image-to-text or sketch-to-code translation.

Transformer Decoder

The Transformer decoder, like the encoder, consists of multiple identical layers. In our model, these decoder layers are designed to work with the encoded representations of the image as well as additional information, such as embedded ground truth caption sequences. The decoder is structured similarly to the encoder but with a few key differences to handle the output generation process. Each decoder block contains three main subcomponents: a masked multi-head self-attention sublayer, a multi-head cross-attention sublayer, and a position-wise feed-forward sublayer. The masked multi-head self-attention mechanism in the decoder ensures that the model can focus on previously generated tokens while predicting the next token in the sequence. This self-attention

mechanism is “masked” to prevent the model from attending to future tokens during training, which enforces autoregressive generation. This ensures that the output is produced sequentially, with each new token being generated based on the previous ones, similar to how natural language is generated in a left-to-right manner. The multi-head cross-attention mechanism allows the decoder to focus on the encoded image embeddings at each time step. By attending to these image representations, the decoder can incorporate relevant visual features into the output sequence. This is particularly useful for tasks such as image captioning or sketch-to-code generation, where the output needs to be grounded in the input image's content. The cross-attention layer allows the model to align the visual features with the tokens being predicted, ensuring that the generated output is accurate and coherent with the input image. The position-wise feed-forward sublayer in the decoder operates similarly to the one in the encoder, applying transformations to each token independently. This step adds further capacity to the model, allowing it to better handle the complexity of generating diverse and detailed output sequences. Additionally, positional encodings are added to the embedded ground truth caption sequences to retain the order of the tokens, ensuring that the sequence of words in the output is consistent with the order of the original text. At each decoding step, the output of the last decoder block is used to predict the next word or symbol in the sequence through a linear transformation, followed by a softmax function. The output dimension of this linear layer corresponds to the size of the vocabulary, allowing the model to predict any word in the vocabulary as the next token in the sequence. This autoregressive process continues until the end of the sequence is reached, and the model has generated a complete output, whether it be a caption, a design description, or another form of structured code.

Attention

Attention can be describe as a similarity between the query and key. They take the form:

$$\text{Attention} = \text{similarity}(q, k) \tag{4-1}$$

Where q represents a query and k represents a key. It's like accessing a database, where we query the database looking for the information we want. To find the similarity between queries and keys, dot product is normally used. It provide the value between 0 and 1. If they are different, obtained value is 0 otherwise it is 1. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Self-Attention

In order to identify dependencies and relationships within input sequences, selfattention

is a method employed in machine learning, especially in natural language processing and CV applications. By taking care of itself, the model is able to recognize and assess the relative value of various input sequence components. In self-attention, vector q, k, v which are actually neural networks (typically linear) have same input (q(x), k(x), v(x)), then they are self- attending.

Scaled Dot-Product Self-Attention

The dimensionality of queries and keys is denoted by d_k, and the dimensionality of values by d_v. The scaled dot-product attention computes the dot product of the queries with the keys, scales the result by the square root of d_k, and applies a softmax function to obtain attention weights. Finally, a weighted multiplication is performed on the data using these attention weights.The complete procedure can be mathematically stated as follows, where Q, K, and V, represent the keys, values, and queries, correspondingly:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4-2}$$

Multi-Head Self-Attention

Instead of performing a single attention function with keys, values and queries, it is more beneficial to linearly project the queries, keys and values h times with different, learned linear projections to dk, dk, and dv dimensions, respectively. By performing the attention function in parallel on each of these projected versions of queries, keys and values, dv –dimensional values are obtained as output. The ability of attending to input from several representation subspaces at different points is provided by multi-head attention to the model.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4-3}$$

Where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Layer Normalization

In deep learning, Layer Normalization (LN) is a technique that helps to stabilize the training process and boost neural network performance. LN separately normalizes each layer's activations for every feature. This means that the activations are scaled and shifted to have a standard normal distribution (mean of 0 and variance of 1) after the mean and variance of the activations are determined independently for each layer.

Position Embedding

Position embedding is used to add spatial information to the image data. Since the

model is actually uninformed of the token's spatial relationship, additional information expressing this relationship can be useful. Usually, this involves assigning tokens weights derived from two high-frequency sine waves, or using a learned embedding. This allows the model to understand that these tokens have a positional relationship. The standard Transformer uses sine and cosine functions of different frequencies:

$$PE(pos, 2i) = \sin(\frac{pos}{10000^{2i}/dmodel}) \tag{4-4}$$

$$PE(pos, 2i+1) = \cos(\frac{pos}{10000^{2i}/dmodel}) \tag{4-5}$$

Where pos denotes the position and i denotes the dimension.

Multi-Layer Perceptron

An MLP is a type of feed forward artificial neural network with multiple layers, including an input layer, one or more hidden layers, and an output layer. It is an Artificial Neural Network in which all nodes are interconnected with nodes of different layers. An input layer, an output layer, and one or more hidden layers with several neurons stacked on top of each other make up a multilayer perceptron. Additionally, neurons in a Multilayer Perceptron can employ any arbitrary activation function, unlike neurons in a Perceptron, which must have an activation function that enforces a threshold, such as ReLU or sigmoid

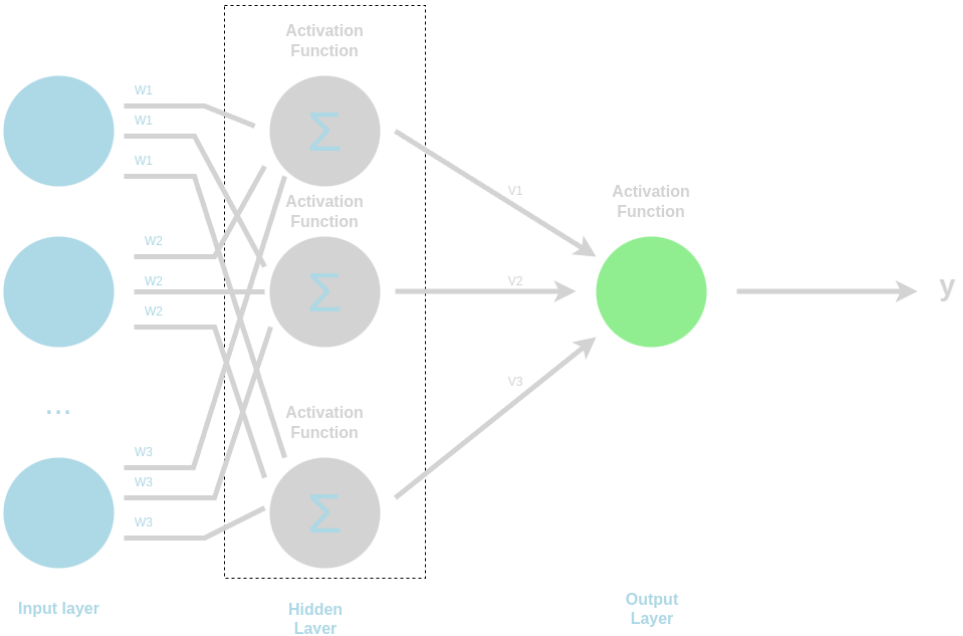

Figure 4-4: Multilayer Perceptron

4.3.4 Domain-Specific Language (DSL)

A DSL is a specialized programming language tailored for a particular task, domain, or set of applications. Unlike general-purpose programming languages such as Python, Java, or C++, which are designed to be versatile and applicable to a wide range of projects, DSLs are focused on a specific problem or field. This specialization allows DSLs to provide more efficient, readable, and expressive solutions within their target domains. In this project, we have design a lightweight and simple Domain-Specific Language to describe the layout and structure of graphical user interfaces (GUIs). The primary goal of our DSL is to allow easy representation and translation of UI designs into code that can be processed by a machine. In this DSL, elements like buttons, text fields, and images will be categorized based on their position, type, and hierarchical relationship within the layout. This hierarchical structure will ensure that the DSL can effectively represent the various components of a GUI in a way that is both simple and intuitive to understand. The key advantage of using a DSL for this project is its simplicity. By narrowing the focus to only the essential components needed to describe the UI, the DSL reduces the complexity of the input space, making it easier for the model to process. This simplification also leads to a smaller vocabulary, as fewer tokens are needed to describe the UI elements. A smaller vocabulary helps improve the efficiency of the model, as it reduces the number of possible options that need to be considered during the learning process. Additionally, to address potential challenges like overfitting and the need for better generalization, we introduce compact convolutional transformers. These models are designed to work efficiently with small datasets by focusing on compact architectures that maintain a balance between performance and model size. Compared to traditional Convolutional Neural Networks (CNNs), compact convolutional transformers are better suited for tasks with limited data, as they have fewer parameters and are less prone to overfitting. This allows the model to learn more robust features from smaller datasets, ensuring better results even when there is limited training data available. The simplicity and specialized focus of the DSL make it a powerful tool for this project, enabling effective and accurate conversion of sketches into functional code. By using a DSL, we can streamline the entire process of UI design representation, making it easier to translate complex visual concepts into structured code while maintaining a high degree of flexibility and efficiency in the overall system.

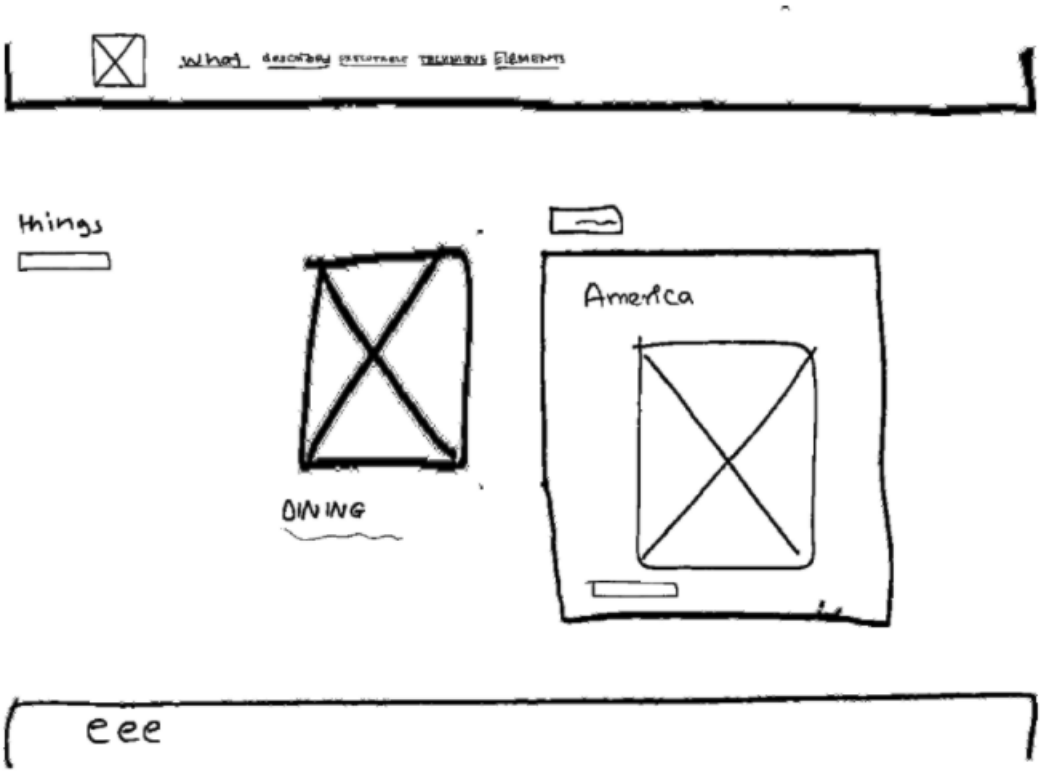

Figure 4-5: Sample Input Image

The sample DSL for above image:

```
header{
    flex{
        logodiv{
            image
        }
        nav{
            navlink
            navlink
            navlink
            navlink
            navlink
        }
    }
}
container{[
    row{
```

```
        div-3{
            text-c
            input
        }
        div-3{
            image
            text
            paragraph
        }
        div-6{
            button
            card{
                text-c
                image
                input
            }
        }
    }
]}
footer{
    text
}
```

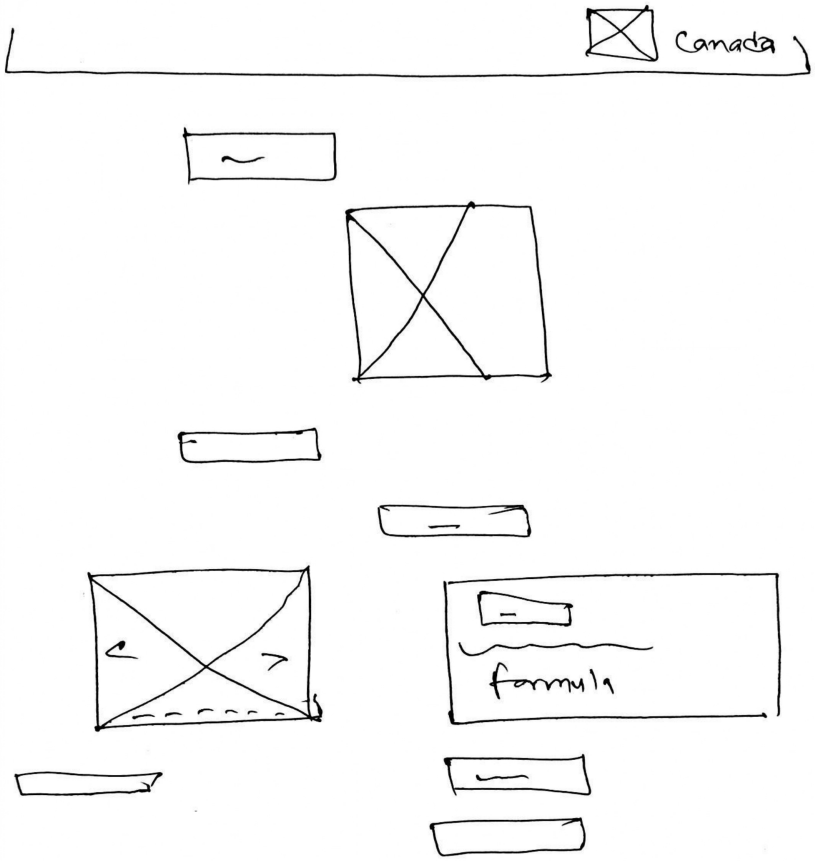

Figure 4-6: Sample Input Image

The sample DSL for above image:

```
header{
    flex{
        logodiv{
            image
        }
        nav{
            navlink
            navlink
            navlink
            navlink
            navlink
        }
    }
}
```

4.3.5 Customization

In this project, simply converting the sketch into HTML code is not enough to create a fully functional web page. While the sketch serves as a basic framework for the layout, it is still lacking essential details such as color schemes, fonts, and other styles. The sketch represents only a rough blueprint of the web page, providing the basic positions of elements but not offering any visual appeal or design choices. To make the final result visually pleasing and user-friendly, the various components on the web page must be styled correctly. Styling not only includes the layout but also focuses on how different elements such as buttons, text, images, and containers appear to the user. This project, therefore, allows for extensive customization options so that the user can personalize the design to their liking. It enables theme selection, where users can choose from a variety of pre-defined themes that reflect different aesthetic styles. These themes determine how the web page will look in terms of color combinations, background designs, and overall visual appeal. Additionally, users have the flexibility to select a color palette that suits their preferences or the brand's color identity. The ability to pick from a range of font styles ensures that the typography aligns with the intended tone and purpose of the web page. Whether it's for a formal business website or a fun and casual blog, the font selection is key in setting the right mood for the page. Apart from theme selection, users can further refine the design by customizing the individual styles of various elements on the page. This includes choosing specific fonts, adjusting font sizes, altering text alignments, and picking colors for headers, paragraphs, buttons, and links. Users can even go as far as deciding how elements like navigation bars, footers, and images should behave and appear on different screen sizes. When it comes to text content, this project will provide randomly generated placeholder text, often referred to as "Lorem Ipsum," which serves as a stand-in for the actual content that the user will later customize. This generated text helps visualize the page layout and provides a placeholder for where the content will go, but users can replace it with their own text at any time. This flexibility allows users to quickly visualize the layout with sample content while giving them the freedom to change it when the real text is available. For images, placeholders are used initially to represent where images will be placed within the design. The user has the option to either provide an image link from the web or upload images directly from their computer. The placeholder image can be replaced easily, allowing users to try out different visuals or insert their own brand images to personalize the page. In summary, this project provides extensive customization options, enabling users to create unique and visually appealing web pages based on their specific requirements. Whether it's choosing a theme, adjusting fonts, altering colors, or adding images and content, the customization feature gives users complete control over the final appearance and design of the web page, making it a flexible and versatile tool for web development.

4.3.6 Compiler

A compiler is a specialized computer program responsible for translating code written in one programming language into another programming language. In our case, the compiler takes the Domain-Specific Language (DSL) that we have created and translates it into HTML and CSS code, which are the standard languages used to design and structure web pages. The role of the compiler is crucial because it bridges the gap between the abstract design defined in DSL and the practical web page that can be viewed in a browser. Our compiler is designed to read a mapping file written in JSON format. This file contains instructions that tell the compiler how each DSL token should be converted into corresponding HTML tags, which are then used to create the structure of the web page. By processing the input DSL code, the compiler produces a structured HTML document, while also applying relevant styles using CSS. The compiler reads these mapping instructions from a JSON file, which acts as an intermediary format that combines both the structural blueprint of the page and the dynamic content provided by the user. This JSON file contains all the necessary data to transform the DSL elements into a fully functional page, complete with user-specific content. Once the JSON file has been generated, the compiler proceeds to convert it into HTML and CSS. The HTML structure is generated based on the hierarchy and relationships defined in the DSL, ensuring that the content is displayed correctly. The CSS, on the other hand, applies styling to the HTML elements, ensuring that the web page looks visually appealing. This process includes applying dynamic variables, such as colors, font choices, and layout styles, which have been selected by the user during the customization phase. The final output is a complete, functional web page that is fully styled and ready to be viewed in a browser. The compiler ensures that the web page retains the structure defined in the DSL, while also integrating the user's content and styling choices. This seamless translation from DSL to HTML and CSS enables users to quickly create customized web pages without the need for manual coding, streamlining the entire development process. In conclusion, the compiler plays a vital role in transforming the abstract design and structure defined by the user in DSL into a fully functional web page. By converting the DSL code into HTML and CSS, the compiler not only generates the necessary content but also ensures that the web page is visually appealing and responsive. With the ability to handle multiple platforms, the compiler offers a versatile solution for creating web pages that can be used across different devices and environments.

4.4 Activation Function

Activation functions play an essential role in neural networks by determining the output of each neuron or unit in the network. They are mathematical functions that take in input

values (the weighted sum of the previous layer's outputs) and apply a transformation to decide the neuron's output. Without activation functions, the neural network would essentially be just a linear regression model, regardless of the number of layers. This is because, when combined, two linear functions still result in a linear function. This limits the ability of the network to learn complex patterns in data, which is why non-linear activation functions are necessary. The purpose of the activation function is to introduce non-linearity into the network. Non-linear functions enable the network to learn from the data more effectively by allowing it to model complex relationships between inputs and outputs. This is crucial because real-world data is often non-linear, and a linear function would not capture the complex relationships that exist in tasks such as image recognition, language translation, or prediction. In this project, the use of specific activation functions helps the neural network to learn and generalize better. Different types of activation functions are used at various layers within the network, and they serve different purposes depending on the architecture and type of network being used. Some functions work better in the hidden layers, while others are more suitable for the output layer.

4.4.1 Softmax Function

The Softmax function is sometimes called the soft argmax function, or multi-class logistic regression since it is a generalization of logistic regression that can be used for 11 multi-class classification. The softmax function is ideally used in the output layer of the classifier where the probabilities are required to define the input images' class. A softmax function calculates the probabilities of each class which the input belongs. The softmax units in the output layer always be equal to number of classes. The probability distribution is different for different classes and the summation value of all probability distribution is 1. The softmax function 4.4 provides the probability values for each classes and class with highest probability value is consider as correct prediction.

$$\text{softmax}\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4-6}$$

Where

z_i : elements of the input vector,

e^{z_i} : standard exponential function applied to each element,

K : number of classes in the multi-class classifier.

The Softmax function is a function that turns a vector of K real values into a vector of

K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the values are converted to values between 0 and 1 by softmax. Thus now they are interpreted as probability. Small or negative inputs are converted to a small probability value.

4.4.2 ReLU Function

Rectified Linear Unit (ReLU) is one of the most widely used activation functions in deep learning and neural networks. It has become a go-to choice due to its simplicity and effectiveness. The ReLU function works by applying a transformation to the input data, where it outputs zero for all negative input values and outputs the input value itself if it is positive. The mathematical equation for the ReLU activation function is:

$$\text{ReLU}(x) = \max(0, x) \tag{4-7}$$

This equation means that if the input,x, is greater than zero, then the output is x. However, if x is less than or equal to zero, the output becomes zero.

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{4-8}$$

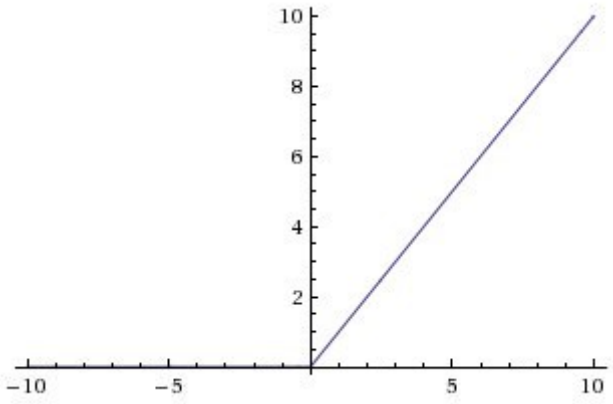

Figure 4-7: Graphical Representation of ReLU Function

4.5 Loss Function and Optimizer

In machine learning, the loss function and optimizer are integral to the learning process. These components directly influence how a model learns from data, improves its performance, and ultimately makes accurate predictions. The loss function is a mathematical formula that calculates the error between the predicted output of the model and the

actual target values. In other words, it quantifies how well or poorly the model is performing. The objective during training is to minimize this loss, effectively improving the model's ability to make accurate predictions. Different types of loss functions are used depending on the type of problem such as classification, regression, or generative tasks. In this project, the loss function will guide the model to generate accurate HTML code from sketch input by minimizing the discrepancies between predicted and actual HTML structures. Once the loss function has been calculated, the optimizer steps in. An optimizer's primary role is to update the model's weights (parameters) in a way that minimizes the loss function. This is done by calculating the gradients, which indicate the direction and magnitude of changes required to reduce the loss. The optimizer adjusts the weights incrementally to steer the model toward a more accurate solution. One of the key aspects of the optimizer is how it controls the learning rate a parameter that determines the size of each step. A proper learning rate is critical to ensure the model does not overshoot the optimal solution or get stuck in a suboptimal one. For the task of converting sketches to HTML code in this project, the optimizer helps fine-tune the parameters, ensuring that the model learns from the sketches efficiently and generates precise code. The combination of a well-chosen loss function and an effective optimizer is fundamental to successful model training. While the loss function guides the model by highlighting the areas where improvement is needed, the optimizer ensures that those improvements are made in the most effective manner possible. Together, they enable the model to learn not just from data but in the way that is most appropriate for generating HTML code from sketches in our project.

4.5.1 Categorical Cross-Entropy

Categorical cross-entropy is one of the most widely used loss functions in machine learning, especially in classification problems where the model is tasked with predicting a class label from multiple possible categories. This function is particularly useful in scenarios where the output involves a probability distribution over different classes, such as in multi-class classification problems. The categorical cross-entropy loss quantifies the difference between the predicted probability distribution and the true distribution, or target values. It does this by penalizing the model more when it assigns higher probabilities to incorrect classes and less when it assigns higher probabilities to correct classes. This helps the model learn to make more accurate predictions over time. The formula for calculating categorical cross-entropy is as follows:

$$\text{Loss} = -\sum_{i=1}^{\text{Output size}} y_i \log(\hat{y}_i) \tag{4-9}$$

Where:

- $\hat{y}_i$ is the predicted probability of the ith class, as output by the model.
- y_i is the true value (or target) for the ith class. This is typically represented as a one-hot encoded vector, where the correct class is marked with a 1 and all others are 0.
- The output size refers to the total number of classes the model is predicting over (the size of the output vector).

In the context of this project, where we aim to predict HTML structures from sketches, categorical cross-entropy would be applied to measure how well the model's output (the predicted HTML tags and structures) matches the target HTML output. By minimizing this loss function, the model learns to improve its predictions over time, eventually generating HTML code that is structurally and syntactically correct, based on the input sketch. The use of categorical cross-entropy is crucial for ensuring that the model's predictions are accurate in a multi-class setting, where each class corresponds to a different HTML element or attribute.

4.5.2 Adam Optimizer

The Adam optimizer, which stands for Adaptive Moment Estimation, is one of the most widely used optimization algorithms in the training of deep learning models. It combines the advantages of two other popular optimization techniques: AdaGrad and RMSProp. Adam computes adaptive learning rates for each parameter by maintaining two moving averages: the first moment m (which represents the mean of the gradients) and the second moment v (which represents the uncentered variance of the gradients). These moving averages are used to adjust the learning rates of the model's parameters in a more informed and dynamic way during training. The key advantage of Adam lies in its ability to adaptively adjust the learning rates for each parameter based on the gradients' first and second moments. This means that parameters associated with larger gradients receive a lower learning rate, while parameters with smaller gradients are updated more aggressively. This adaptive approach helps accelerate convergence, particularly in problems involving sparse gradients or noisy data. Moreover, Adam is designed to work well with non-stationary objectives, making it highly effective in real-world applications where data may change or evolve over time.

The optimization process in Adam involves the following steps:

- Compute the gradient of the loss function with respect to the model parameters.

- Update the first moment estimate (m) as the exponential moving average of the gradient.
- Update the second moment estimate (v) as the exponential moving average of the squared gradient.
- Compute a bias-corrected estimate of the first and second moments to counteract initialization bias.
- Adjust the learning rate for each parameter based on the moving averages of the gradients and their variances.
- Update the model parameters using these adjusted learning rates.

Adam has gained popularity due to its simplicity of implementation, computational efficiency, and low memory requirements. It does not require a lot of hyperparameter tuning, making it user-friendly for both beginners and experienced practitioners. Additionally, Adam is invariant to diagonal rescaling of the gradients, meaning that it is robust to changes in the scale of the input data, a property that makes it suitable for a wide range of deep learning tasks. In the context of this project, where the task involves generating HTML structures from input sketches, Adam provides an efficient way to train the model while adapting the learning rates based on the gradients of the loss function. Its ability to handle large datasets, combined with the adaptive learning rate mechanism, helps the model converge faster and achieve better performance. By using Adam, we ensure that our model is both computationally efficient and capable of adapting to the complexities of the task at hand.

4.6 Performance Metrics

In machine learning, performance metrics are essential for evaluating how well a model performs on a given task. These metrics offer quantitative measures of a model's accuracy and reliability, guiding the development process by helping to determine whether the model is underfitting, overfitting, or generalizing well to new data. For text generation tasks, such as generating HTML from sketches, using appropriate metrics is crucial to assess the quality and relevance of the generated content compared to the expected output. For this project, we utilize two popular performance metrics—BLEU (Bilingual Evaluation Understudy) and ROUGE (Recall-Oriented Understudy for Gisting Evaluation)—to evaluate the model's effectiveness. Both of these metrics are commonly used in natural language processing (NLP) tasks, especially in machine translation and text summarization, but they can also be applied to evaluate the quality of generated text in our use case.

4.6.1 BLEU

Bilingual Evaluation Understudy (BLEU) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. This is a common metric used in machine translation tasks, which seeks to measure how closely a machine-generated text resembles what a human would have generated, given the same input. It is based on n-gram based precision. Four sub metrics are denotes as BLEUn, for n = 1, 2, 3, 4. For a candidate sentence a and a set of reference sentences b, the BLEU score is calculated as:

$$\mathrm{BLEU}_n(a,b) = \frac{\sum_{w_n \in a} \min\left(\mathrm{count}_a(w_n), \max_{j=1,\ldots,|b|} \mathrm{count}_{b_j}(w_n)\right)}{\sum_{w_n \in a} \mathrm{count}_a(w_n)} \tag{4-10}$$

where wn denotes n-gram, counta(wn) denotes count of n-gram wn in sentence

4.6.2 ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is an evaluation metric specially designed for automatic summarization that can be used for machine translation. It measures the overlap of n-grams between the generated sequence and reference sequence. The ROUGE score is calculated as:

$$\text{ROUGE-N Precision} = \frac{\sum_n \mathrm{count}_{\mathrm{match}}(n)}{\sum_n \mathrm{count}_{\mathrm{generated}}(n)} \tag{4-11}$$

$$\text{ROUGE-N Recall} = \frac{\sum_n \mathrm{count}_{\mathrm{match}}(n)}{\sum_n \mathrm{count}_{\mathrm{reference}}(n)} \tag{4-12}$$

$$\text{ROUGE-N F1} = \frac{2 \times \text{ROUGE-N Precision} \times \text{ROUGE-N Recall}}{\text{ROUGE-N Precision} + \text{ROUGE-N Recall}} \tag{4-13}$$

4.7 Flowchart

4.7.1 During Training

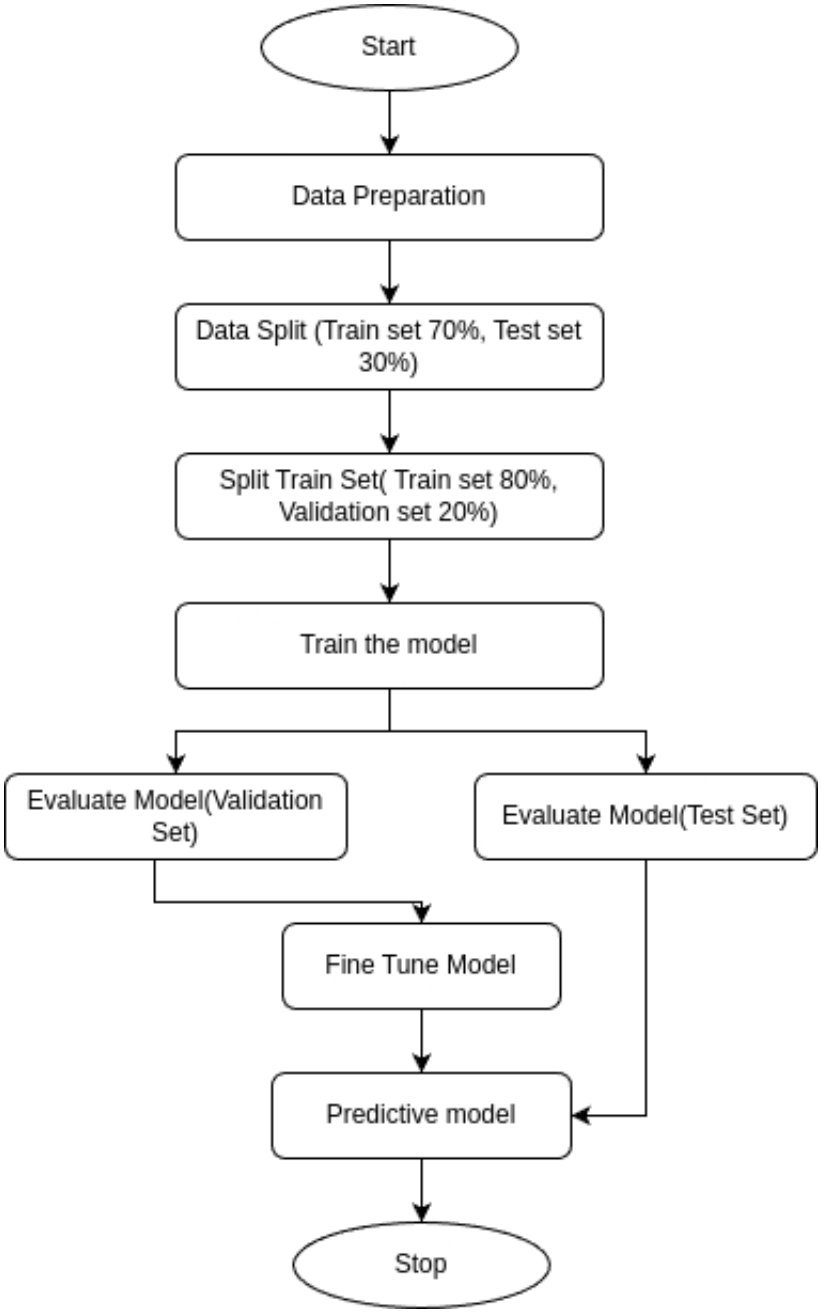

Figure 4-8: Training Flowchart

4.7.2 During Testing

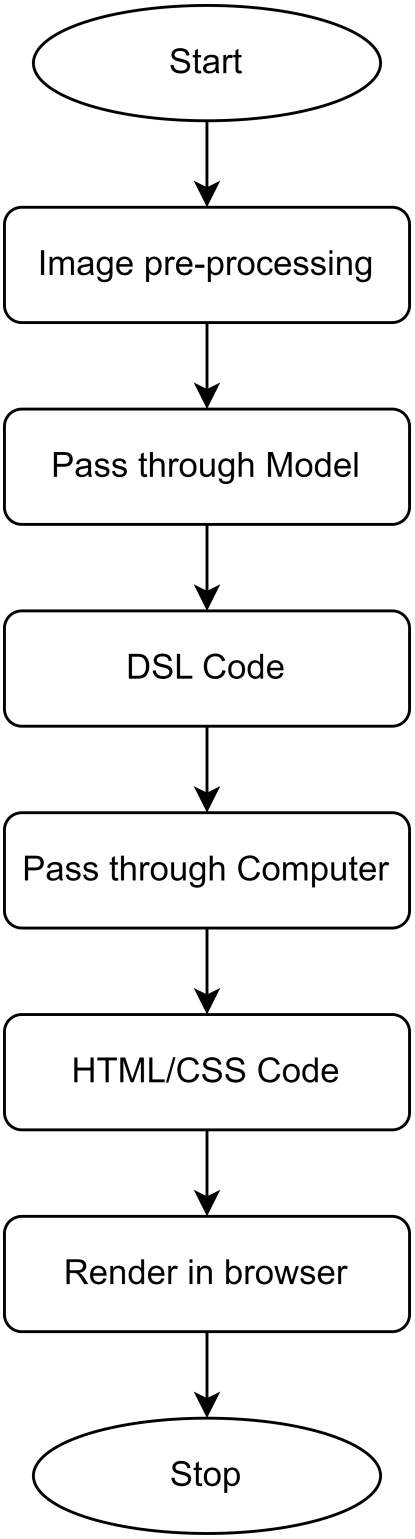

Figure 4-9: Testing Flowchart

5 IMPLEMENTATION DETAILS

5.1 Dataset Generator Implementation

Initially, we collected the multiple number of hand-drawn sketches for the each element under consideration. Our focus was on selecting elements that provided the most efficient and informative structural layout for the user Interface. Once we finalized the core set of elements, the individual sketches are combined to create a many numbers of sketch wireframe required to train our Transformer model. This approach allows us to create a larger and diverse dataset that contains different design possibilities and layout variations.

5.1.1 Steps taken in Dataset Generation

Random DSL generator

The first step in the data generation process is the creation of the Domain-Specific Language (DSL) code. This DSL code was generated using a set of custom-made rules designed to ensure effective representation of web page layouts. The rules were created to allow different combinations of elements, ensuring a variety of layout that represents the User Interface. Each generated DSL code was unique, with varying arrangements of UI components so that it provides the broad spectrum of design possibilities. The DSL generation process involves refinement and testing to be sure that the generated code was both meaningful and represents the real-world web layouts.

Compiling the DSL Code

Once the DSL code was generated, the next step is to compile or map it to the corresponding HTML code. This process involved translating the high-level DSL representation into a concrete and functional HTML structure which can be used by the browers to give User Interface. During the compilation phase, special attention was given to maintaining semantic correctness and ensuring that the generated HTML structure follows rules of design of web page and aligned with best practices in web development. The compiled HTML code served as the foundation upon which additional styling and interactivity were layered.

Rendering the produced HTML Code

After compiling the DSL code into HTML, the resulting web pages were rendered in a Chrome browser. During the rendering process, a specialized CSS file was applied to style the elements, ensuring a visually distinct and well-structured layout is obtained. Different elements within the page were rendered with specific background colors with the help of specialized CSS, which played a crucial role in subsequent pro-

cessing steps. Additionally, a JavaScript file was used to dynamically assign random height, width, and positioning values to each element, introducing variability and making the dataset more comprehensive. Once the elements were properly rendered and positioned, screenshots of the pages were captured for further processing.

Finding the outline of the different element

To identify and extract the individual elements from the rendered pages, a contour detection technique is used. This involved filtering out the background colors assigned during the rendering phase and using image processing techniques to detect the contours of each UI component. Through this approach, the precise dimensions and positions of various elements were obtained. These bounding boxes provides critical information which is later used to align the hand-drawn sketches with the corresponding UI components.

Placement of hand-drawn sketch

Once the bounding boxes for each element are identified, the next step is to overlay the hand-drawn sketches onto the detected elements. To achieve this, the aspect ratios of the hand-drawn sketches are compared with those of the bounding boxes. The sketches are then sorted based on the closest matching aspect ratios to ensure an optimal fit. From the top-ranked sketches, a random selection of up to eight sketches is made, and one of them is placed within the corresponding bounding box. This step ensures a realistic and visually appealing integration of the sketches within the generated layouts.

Rules for creating DSL

To ensure consistency and structure in the generated web layouts, a set of rules and guidelines is established for DSL generation. The structure was based on a hierarchical representation of webpage elements, where each key element had a predefined set of possible child elements. For example, the 'root' element could contain 'header', 'container', and 'footer' sections, while a 'row' could include various sizes of 'div' elements. The layout generation also incorporated a 12-column grid system, which is widely used in responsive web design. This grid-based approach allowed for flexible and adaptive layouts while maintaining a sense of order and balance. Furthermore, constraints are introduced to govern the occurrence and arrangement of elements. These constraints defined the minimum and maximum number of occurrences for each component, the order in which child elements should appear, and the allowable combinations for dividing rows into columns. By enforcing these rules, we ensured that the generated layouts were both diverse and logically structured. The flexibility provided by these constraints allowed for a wide range of layout variations while maintaining essential structural integrity. Overall, the structured approach for dataset generation enable us to create a rich

and diverse set of training data that captured a wide variety of design possibilities. This process laid the groundwork for training our Transformer model effectively, ensuring it could generalize well across different UI layouts and design elements.

5.2 Model Implementation

We have developed a transformer-based architecture designed to convert sketch images into DSL code. The architecture consists of an encoder that processes the input images (sketches) and a decoder which is responsible for generating the corresponding DSL code. To achieve high accuracy and efficiency, our model leverages Convolutional Neural Networks (CNNs) for feature extraction from images and transformer layers for sequence modeling. This combination allows the model to effectively translate visual representations into structured textual output, which is the DSL code.

5.2.1 Input Processing

The model is designed to accept two types of inputs for the training of the model. This is the processing that is done in the training phase. Otherwise, we just have to input the image:

i. **Image Input** The first input to the model is a grayscale sketch image with dimensions (None, 850, 600, 1). Each sketch serves as a visual representation of a UI layout and is processed through the CNN layers to extract meaningful spatial features and patterns.

ii. **Text Input** The second input is the textual representation of the DSL code associated with the given sketch. It has a shape of (None, 120), where each sequence represents the structural layout and element relationships defined in the DSL format. This input acts as a ground truth reference during the training process.The input processing pipeline involves several key steps, including image normalization, tokenization of the DSL text input to enable efficient processing by the transformer layers. By integrating both image and text inputs, the model is trained to establish a strong correlation between visual layouts and their corresponding code representations, which provides the accurate and reliable predictions.

5.2.2 Convolutional Tokenizer

The Convolutional Tokenizer is the first critical component of our transformer-based architecture, which is responsible for processing the input sketch images and transforming them into meaningful feature representations. It helps in focusing on the feature that are important for the DSL generations so, we do not require large number of input images

to train the transformer model. This custom layer contains a series of convolutional and pooling operations to convert the 2D grayscale image into a sequence of feature vectors. The output of the tokenizer has a shape of (None, 513, 128), which indicates that the input image is transformed into 513 tokens, each represented by a feature vector of 128 dimensions. This transformation is crucial as it prepares the data for further processing by the subsequent transformer layers. This layers encode the features present in the image into a feature vectors.

Convolutional Layers

The Convolutional Tokenizer comprises multiple convolutional layers (Conv2D), each of which are designed to extract hierarchical features from the input image. The number of filters increases as the network goes deeper, which helps in richer and more abstract representation of the image features. Each convolutional layer is followed by zero-padding and max-pooling operations, which help in preserving spatial dimensions and capturing essential patterns while also reducing computational complexity. Dropout layers are also placed within the network to mitigate the risk of overfitting and improve generalization.

Layer-wise Breakdown

i) **First Convolutional Layer:**

- Filters: 32
- Kernel Size: 7x7
- Stride: 1
- Padding: None initially, followed by ZeroPadding2D of 1 to preserve spatial dimensions.
- Activation Function: ReLU
- Pooling: MaxPooling2D with a window size of 3x3 and a stride of 2.

In this initial layer, the large kernel size of 7x7 is used, which allows the model to capture broader structural features from the sketch image. The addition of zero-padding ensures that the spatial dimensions remain intact, while max-pooling helps in reducing the feature map size and computational load.

ii) **Second Convolutional Layer:**

- Filters: 64
- Kernel Size: 5x5

- Stride: 1
- Padding: None initially, followed by ZeroPadding2D of 1.
- Activation Function: ReLU
- Pooling: MaxPooling2D with a 3x3 pooling window and stride of 2.
- Dropout: Applied with a rate of 0.1.

The second layer increases the depth of feature maps, allowing the model to learn more complex patterns. Dropout is applied after this layer to prevent overfitting by randomly deactivating some neurons during training, encouraging the network to learn more robust features.

iii) Third Convolutional Layer:

- Filters: 64
- Kernel Size: 3x3
- Stride: 1
- Padding: None initially, followed by ZeroPadding2D of 1.
- Activation Function: ReLU
- Pooling: MaxPooling2D with a 3x3 pooling window and stride of 2.

With a smaller kernel size of 3x3, the third layer focuses on detecting finer details in the image while maintaining an efficient processing pipeline through down-sampling via max-pooling.

iv) Fourth Convolutional Layer:

- Filters: 128
- Kernel Size: 3x3
- Stride: 1
- Padding: None initially, followed by ZeroPadding2D of 1.
- Activation Function: ReLU
- Pooling: MaxPooling2D with a 3x3 pooling window and stride of 2.
- Dropout: Applied with a rate of 0.1.

The fourth layer further enhances the feature representation by increasing the

number of filters to 128. This allows the model to detect complex patterns, edges, and finer textures. The dropout layer once again aids in generalization.

v) **Fifth Convolutional Layer:**

- Filters: 128
- Kernel Size: 3x3
- Stride: 1
- Padding: None initially, followed by ZeroPadding2D of 1.
- Activation Function: ReLU
- Pooling: MaxPooling2D with a 3x3 pooling window and stride of 2.

The final convolutional layer retains the same number of filters as the previous layer, ensuring consistency in feature extraction and maintaining high-dimensional feature representations. The resulting feature maps undergo max-pooling to down-sample and prepare them for the subsequent flattening process.

Flattening the Output

After passing through all convolutional and pooling layers, the output feature maps are flattened into a shape of (513, 128). This transformation converts the spatially distributed feature maps into a sequence of 513 tokens, each represented by a 128-dimensional feature vector. This sequential representation is then fed into the transformer model for further processing and translation into DSL code. The Convolutional Tokenizer plays a crucial role in bridging the gap between raw image input and the transformer-based sequence processing model. Its structured design ensures efficient feature extraction, dimensionality reduction, and effective preparation of input data for downstream tasks. By incorporating a combination of convolutional layers, pooling operations, and dropout mechanisms, our tokenizer ensures high performance, robustness, and scalability for processing sketch images of varying complexity and style.

5.2.3 Positional Embedding

After the convolutional tokenization process, positional encoding is applied to the token embeddings to incorporate spatial information. Since transformers do not inherently capture the order or spatial relationships of input sequences, the use of positional encoding is essential for enabling the model to understand the positional relationships between different parts of the image.

We have used sinusoidal positional encoding, which is a widely used technique that pro-

vides a unique representation for each position in the input sequence. The sinusoidal encoding functions are designed to produce continuous and periodic values, which allow the model to generalize to sequences of varying lengths and effectively capture patterns in spatial arrangements.

Sinusoidal Positional Encoding
This encoding scheme allows each position to have a unique representation based on sine and cosine functions of different frequencies, enabling the model to distinguish between different token positions efficiently.

Combining Tokenized Features with Positional Encoding
Once the token embeddings are obtained from the convolutional tokenizer, the positional encodings are added element-wise to the feature vectors. This addition allows the model to retain positional context alongside the extracted features, enhancing its capability to understand spatial dependencies in the sketch images. The integration of positional embeddings is crucial in ensuring that the model effectively captures both local and global structures within the image layout, which is fundamental for accurate DSL code generation. The combined representation is then passed to the transformer encoder for further processing, where self-attention mechanisms utilize both feature and positional information to derive meaningful relationships among different elements.

5.2.4 Encoder

The encoder in our model consists of three transformer blocks, each of which are designed to process the tokenized image data efficiently. Every block in the encoder follows a structured approach that includes several critical components: layer normalization, multi-head attention (self-attention), residual connections, and a Feed-Forward Neural Network (FFNN). The encoder is responsible for transforming the input image tokens of shape (513x128) into a meaningful representation that the decoder can further utilize for DSL code generation. The output of each encoder block maintains the same shape of (513x128), preserving spatial consistency while enriching feature representation.

Components of the Transformer Encoder

i) Layer Normalization

- Each layer normalization operation consists of 256 trainable parameters and serves to normalize inputs across the 128 feature dimensions.
- This normalization step is crucial for improving training stability by reduc-

ing internal covariate shifts, allowing the model to converge faster and more efficiently.

ii) Multi-head Attention (Self-attention)

- The multi-head attention mechanism is a key component in capturing dependencies across different token positions within the sequence.
- Each multi-head attention layer has 329,728 parameters, contributing significantly to the model's ability to learn complex relationships.
- We have employed 5 attention heads, which allows the model to focus on multiple aspects of the input simultaneously.
- The input and output dimensions for the attention layer remain consistent at (513x128), ensuring no loss of spatial or feature-based information.

iii) Residual Connections

- To facilitate gradient flow and prevent vanishing gradients, residual connections are used.
- These connections add the input of the attention layer (513x128) directly to its output, effectively bypassing the complex transformations and ensuring that important lower-level features are preserved throughout the layers.

iv) Feed-forward Neural Network (FFNN)

The feed-forward network within each transformer block is composed of two dense layers:

(a) **First Dense Layer:**

- Has a shape of (513x128) and consists of 16,512 parameters.
- This layer applies linear transformations to enrich the feature representation.

(b) **Dropout Layer:**

- A dropout layer is applied after the first dense layer to prevent overfitting and improve generalization during training.

(c) **Second Dense Layer:**

- Another dense layer of the same shape (513x128) with 16,512 parameters is applied.
- This is followed by another dropout layer to further regularize the model.

Each of the three transformer encoder blocks follows the above structure, progressively refining the image token embeddings while maintaining the original spatial dimensions. This multi-layered processing enables the model to capture both local and global dependencies within the sketch image representations, which is critical for accurate DSL code generation.

5.2.5 Decoder

The decoder is more complex. It is designed to generate DSL code sequences from the encoded image representations. It consists of four transformer blocks, each contributing to the sequential processing of the input embedding while attending to the encoder's output. The decoder processes an input sequence of shape (120x128) and interacts with the encoder output of shape (513x128) to produce the final output sequence. Each block in the decoder includes:

i Self-attention Layer

The self-attention mechanism in the decoder is crucial for processing the input sequence by allowing each position to attend to all previous positions. This mechanism ensures that predictions are made based only on preceding tokens, preventing information leakage from future tokens.

- The self-attention layer consists of 329,728 parameters, making it a key computational component.
- The input and output dimensions of this layer are (120x128).
- A masking mechanism is applied to ensure autoregressive decoding, allowing the model to focus solely on past and current tokens.

ii Cross-attention Layer

The cross-attention layer enables the decoder to focus on relevant parts of the encoder's output while generating the target sequence. This layer establishes dependencies between the input image tokens and the generated code sequence.

- It contains 329,728 parameters, similar to the self-attention layer.
- The layer takes the encoder's output of shape (513x128) as input and processes it alongside the decoder's sequence of shape (120x128).
- This mechanism allows the decoder to align each generated token with specific image features.

iii Feed-forward Neural Network (FFNN)

The feed-forward network is used to further refine the features extracted by the

attention layers. It consists of two dense layers, each followed by a dropout layer to prevent overfitting and enhance generalization.

- Each dense layer has a size of (120x128) and consists of 16,512 parameters.
- The first dense layer applies linear transformations, followed by a dropout layer to introduce regularization.
- The second dense layer, also followed by dropout, further refines the output before passing it to the next decoder block.

iv Layer Normalization and Residual Connections

Layer normalization ensures stable training by normalizing activations across the feature dimensions, which helps in maintaining consistent gradient flow.

- Each layer normalization contains 256 parameters, applied across the 128 feature dimensions.
- Residual connections play a crucial role by adding the input of shape (120x128) to the output of each sublayer, preserving important information and improving gradient flow.

Each of the four decoder transformer blocks processes the input embeddings while interacting with the encoder output, gradually refining the generated sequence. With self-attention, cross-attention, feed-forward networks, layer normalization, and residual connections, the decoder is well-equipped to produce accurate DSL code based on the provided sketch image representation.

5.2.6 Attention Mechanism

The attention mechanism is a crucial component of our model, facilitating the processing of both input image tokens and the generated DSL code sequences. The model employs several multi-head attention layers, each designed to capture dependencies across the tokenized representations effectively.

i) Attention in the Encoder

In the encoder, the attention mechanism consists of self-attention layers, which enable each token within the image representation to attend to all other tokens. This allows the model to capture intricate relationships between different parts of the sketch, ensuring a rich feature representation.

- Self-Attention: Each token in the sequence of shape (513x128) can interact with every other token, leading to a comprehensive understanding of the entire image layout.

- Multi-Head Attention Multiple attention heads allow the model to focus on different aspects of the image simultaneously, improving its ability to discern patterns and spatial relationships.

ii) **Attention in the Decoder**
The decoder contains two types of attention mechanisms: self-attention and cross-attention, which work together to generate the target sequence effectively.

- Self-Attention in Decoder: The decoder's self-attention layers process the sequence of shape (120x128), enabling each token to attend to previous tokens. This ensures that each predicted token is generated based only on previously seen information, maintaining the autoregressive nature of the sequence generation.

- Cross-Attention The cross-attention layers allow the decoder to focus on relevant parts of the encoder's output while processing the generated sequence.This mechanism aligns the input sketch features (of shape 513x128) with the generated sequence, ensuring meaningful code generation.

iii) **Multi-Head Attention** Each attention layer employs a multi-head mechanism to improve learning by attending to different aspects of the sequence in parallel.

- The number of attention heads used in our model is 5, which enhances the model's capability to capture diverse relationships.

- Each multi-head attention layer in the encoder and decoder contains 329,728 parameters, contributing significantly to the model's complexity and learning power.

- The attention layers have consistent input and output dimensions, preserving spatial integrity.

5.2.7 Masking

Masking is a crucial technique which is used in the decoder's self-attention layers to ensure proper sequential processing during the generation of the output sequence. The model implements causal masking, which plays an essential role in autoregressive generation by preventing tokens from accessing future information.

Causal Masking in Self-Attention
Causal masking ensures that each position in the output sequence can only attend to previous positions. This is achieved by applying a mask over the attention weights, effectively setting the attention scores of future positions to negative infinity. It prevents

information flow from future tokens.

- **Purpose:** It maintains the sequential nature of generation, ensuring that the model predicts each token based solely on previously generated tokens.
- **Implementation:** During the self-attention computation, an upper triangular matrix filled with negative infinity values is added to the attention scores, effectively masking out future positions.
- **Impact:** It helps the model avoid data leakage and ensures that predictions are conditioned correctly, improving the accuracy and coherence of generated sequences.

Types of Masking Used

- **Causal Masking:** Applied in the decoder's self-attention to maintain autoregressive properties.
- **Padding Masking:** Optionally used to ignore padding tokens in sequences, ensuring that padding elements do not contribute to the attention mechanism.

Advantages of Masking

- **Prevents Information Leakage:** Ensures the model adheres to the sequential generation requirement.
- **Improves Training Stability:** Helps the model learn dependencies in the correct order.

By using causal masking in the model, the model can generate the target sequence in an orderly fashion, where each token is predicted based only on past information. This mechanism is crucial for achieving high-quality autoregressive sequence generation in the context of sketch-to-code translation.

5.2.8 Output Layer

The final stage of our model is the output layer, which is responsible for generating the predicted DSL code tokens based on the processed input features. This layer plays a critical role in transforming the decoder's learned representations into meaningful output sequences.

Structure of the Output Layer:

- The output layer consists of a dense (fully connected) layer with 31 units, which corresponds to the vocabulary size of the output tokens.
- The vocabulary size is determined by the number of possible tokens used in the DSL code, minus one to account for the special start token used during sequence generation.
- Each unit in the dense layer represents a potential token, and the layer outputs raw logits indicating the likelihood of each token at every position in the output sequence.

Functionlaity

i) Logits Generation

- The dense layer produces logits for each token position in the sequence, representing the unnormalized probabilities of each token in the vocabulary.
- These logits are then processed by a softmax function to convert them into probability distributions over the vocabulary.

ii) Token Prediction

- The highest probability token is selected at each position to construct the final sequence.
- This sequence is then used to generate the DSL code corresponding to the input sketch.

Training Considerations

- During training, the model uses teacher forcing, where the correct tokens are provided at each step to guide learning.
- Cross-entropy loss is used to optimize the output layer by comparing predicted tokens with ground truth tokens.
- Dropout techniques and regularization strategies are applied to prevent overfitting and ensure robust generalization.

The output layer is a crucial component that bridges the decoder's learned representations and the final code output. By producing logits for each token in the vocabulary, the model can effectively generate accurate DSL code representations based on the provided sketch inputs.

5.3 Model Complexity

The designed transformer-based model consists of a total of 3,488,447 trainable parameters, striking a balance between model capacity and computational efficiency. This parameter count reflects the intricacies of the architecture, ensuring that the model can effectively capture the complexity of the sketch-to-code translation task while remaining feasible for training and deployment.

Breakdown of Model Complexity

i. **Encoder Complexity:**
   - The encoder comprises multiple convolutional and transformer layers, contributing a significant portion of the total parameter count.
   - Each convolutional layer extracts hierarchical spatial features from the input sketches, while transformer layers provide contextualized representations.

ii. **Decoder Complexity:**
   - The decoder consists of several transformer blocks, which include self-attention and cross-attention mechanisms.
   - These layers allow the decoder to generate coherent DSL code sequences by attending to both past outputs and encoder outputs.

iii. **Attention Mechanisms:**
   - Multi-head attention layers play a crucial role in both encoder and decoder by enabling the model to focus on different aspects of the input.
   - With five attention heads, the model can efficiently capture relationships between tokens.

iv. **Feed-forward Networks:**
   - Fully connected layers within both the encoder and decoder add to the parameter count, enhancing the representational power of the model.
   - These layers contribute to non-linear transformations crucial for feature abstraction.

Optimization Considerations

- Despite the substantial number of trainable parameters, the model is designed to optimize memory usage and computational efficiency.

- Techniques such as dropout and weight regularization are employed to prevent overfitting while maintaining generalization capabilities.

The model achieves an optimal balance between complexity and efficiency, ensuring it can process intricate sketch inputs and generate high-quality DSL code within reasonable computational constraints. With 3,488,447 trainable parameters, it is well-suited for the demands of real-world applications.

5.4 Vocabulary of Model

The DSL (Domain-Specific Language) used in this project consists of a total of 32 different symbols and words, which serve as the fundamental building blocks for generating structured code from sketch inputs. These elements include both functional commands and structural identifiers essential for the DSL code representation.

Special Symbols

The vocabulary includes three special symbols that play critical roles in sequence processing:

- <**Start**> **Symbol:**Assigned the value 31, this token marks the beginning of a sequence and helps the model recognize the start of generation.
- <**End**> **Symbol:**Signals the end of a sequence, indicating when the model should stop generating output.
- <**Pad**> **Symbol:**Assigned the value 0, this token is used to pad shorter sequences to ensure uniform input dimensions, enabling efficient batch processing.

Textual Representation and Vector Conversion

To process the DSL code efficiently, the textual representation is converted into a numerical vector form. Each unique word or symbol within the vocabulary is assigned a unique integer value ranging from 0 to 31, ensuring a one-to-one mapping between textual and numerical representations.

Embedding Layer

Once the textual representation is transformed into numerical form, it is passed through an embedding layer, which converts the one-dimensional token vectors into high-dimensional dense vectors. These dense vectors have a size corresponding to the chosen projection dimension, allowing the model to capture semantic relationships between different tokens.

Importance of Vocabulary Encoding

- **Uniformity:** The padding symbol ensures all input sequences have consistent lengths, which simplifies model training and inference.
- **Sequence Control:** The `<Start>` and `<End>` symbols provide clear boundaries for the generated sequences, improving accuracy.
- **Efficient Processing:** The numerical encoding enables rapid processing and compatibility with neural network operations.

The well-structured vocabulary of 32 symbols, including special tokens, allows the model to process input DSL sequences effectively and ensures that the generated code adheres to the defined DSL syntax and structure.

5.5 Output DSL

Our transformer model generates DSL (Domain-Specific Language) code as the output, which primarily focuses on representing the structural elements present in the given web page. The generated DSL output provides an abstract layout of the webpage by defining the arrangement and types of elements but does not include the actual content such as text, images, or Uniform Resource Locator (URL) links. These content elements must be manually added or processed in subsequent steps to achieve a fully functional webpage.

Characteristics of the Generated DSL

- **Structural Representation:**
  - The DSL output specifies the placement and type of elements (e.g., divs, headers, footers, buttons, and forms) without embedding actual content.
  - The generated code serves as a skeletal framework to be enriched with further details.
- **Content Exclusion:**
  - Text content, image sources, and URLs are not part of the generated output.
  - Additional data integration is required post-generation to make the web page functional.

5.6 User Interface

The user interface (UI) provides a platform for users to customize the style and content of the generated web page. Through the interface, users can modify various aspects of the web elements. User can enhance the visual appeal and ensuring the content aligns with their requirements.

Customizable Elements

Users can make changes to the following aspects of the page:

- **Text:**
  - Modify textual content within headers, paragraphs, and buttons.
  - Change font styles, sizes, and weights.
- **Images:**
  - Upload new images or replace existing ones.
  - Adjust image size and alignment.
- **Links:**
  - Update hyperlink destinations.
  - Modify link styles such as color, underline, and hover effects.
- **Color:**
  - Change background and foreground colors of different elements.
  - Customize color schemes for various components.
- **Font Size and Spacing:**
  - Adjust font sizes for improved readability.
  - Modify spacing between elements to optimize layout.

Message Handling and Compilation

All changes made through the interface are translated into a structured JSON format, which is then passed to the compiler for processing. This ensures that user preferences are accurately reflected in the final rendered web page.

5.7 DSL to JSON compiler

First, we convert the generated DSL output into the JSON format by adding relevant information into it. Text, image, URL are added in this process which is obtained from

the user through the User Interface. It is converted using the DSL mapping file which contains all the information related to the DSL element. Generated JSON file can be used to generate the required HTML and CSS. JSON also contains the style required to customize the color based on the user input. All the data required for the webpage is passed through the JSON file.

5.8 JSON to HTML compiler

JSON file obtained from the JSON compiler consist all the information required to convert it into the required HTML and CSS file. Each JSON file is mapped according to the DSL mapping file and according to the hierarchy of the element, it is render to get HTML file. CSS file is created using the dynamic CSS variable which helps in inserting the requirement of the user from the JSON file. We, then get the required html file with its corresponding style in the CSS file.

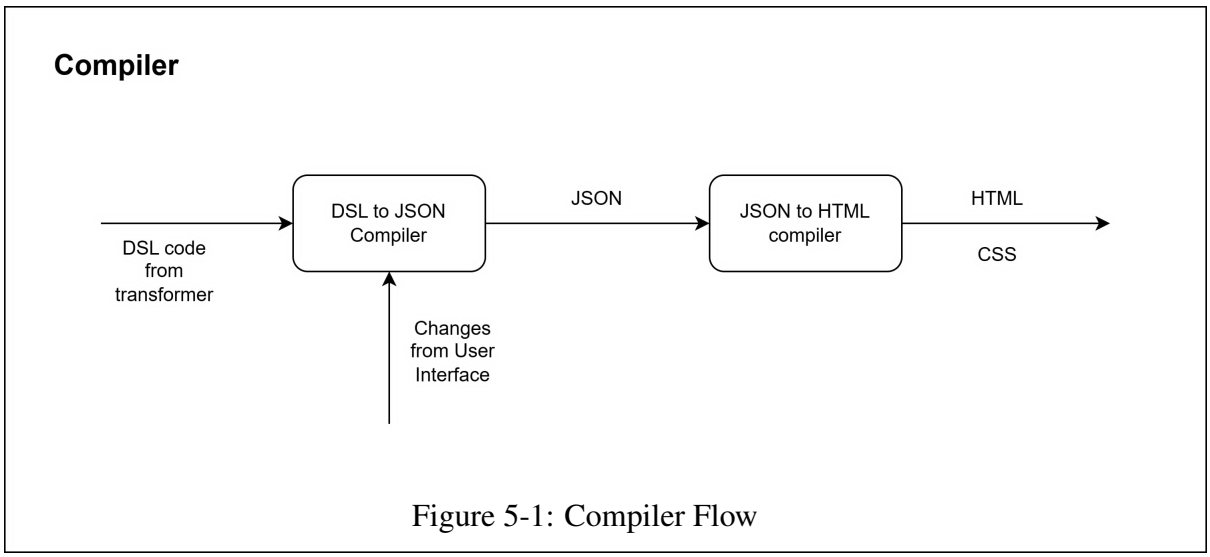

Figure 5-1: Compiler Flow

5.9 Training Details

The model is trained using supervised learning with paired sketch-code data. It employs teacher forcing, where ground truth tokens are fed to the decoder during training to accelerate convergence and improve accuracy.

Objective Function

The training goal is to minimize cross-entropy loss between predicted and actual tokens, guiding the model to generate accurate DSL code sequences.

Optimization

Both convolutional and transformer layers are optimized simultaneously, ensuring effective feature extraction and mapping to DSL code. The training process uses super-

vised learning, teacher forcing, and data augmentation to enhance accuracy and generalization in DSL code generation.

5.10 Testing Details

During testing, the model generates DSL code, predicting one token at a time. It processes the input sketch and sequentially predicts tokens, using previously generated tokens as input for subsequent predictions.

Evaluation Metrics

The model's performance is assessed using the following metrics:

- **BLEU (Bilingual Evaluation Understudy):** Measures the similarity between the generated code and the reference code by comparing n-grams.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** Evaluates the recall aspect by measuring the overlap of word sequences between the generated and reference codes.

Human Evaluation

In addition to automated metrics, human evaluation is conducted to assess the quality and relevance of the generated DSL code. Experts review the code to ensure it aligns with the intended design and accurately represents the input sketch.

6 RESULT AND ANALYSIS

6.1 Dataset 1

6.1.1 Model Training

The model is trained for 8 epochs taking batch size of 32. The training set consists of 8951 samples and testing set consists of 2238 samples. All these samples are generated using random dataset generator.

6.1.2 Loss vs Epoch Graph

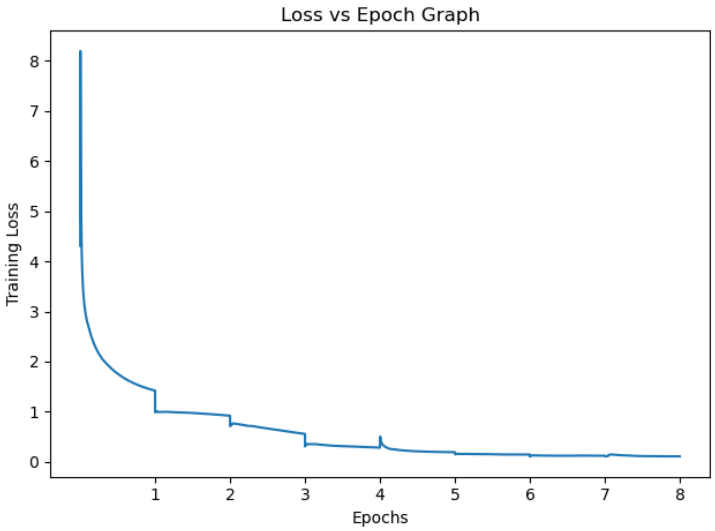

Figure 6-1: Loss vs Epoch graph for training time

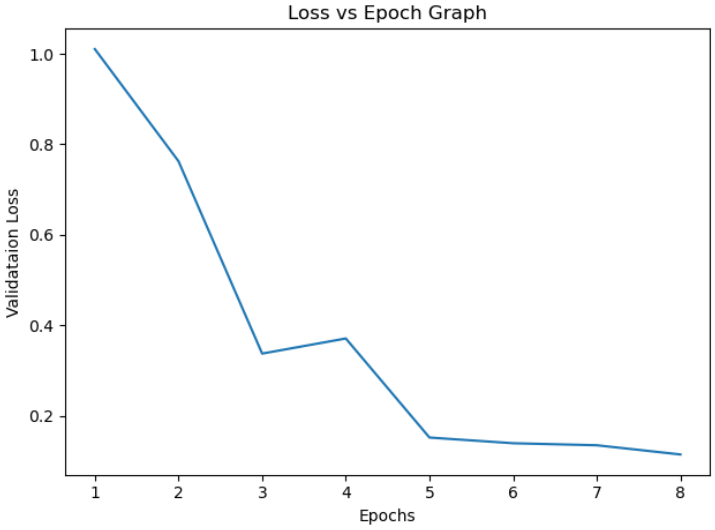

Figure 6-2: Loss vs Epoch graph for validation time

Initially, the loss for the model is very high. In first epoch, the loss gradually decreased and reached around 1.5. In the following epochs, the loss decreased steadily and reached to 0.1 in the final epoch. The loss in validation set also remain comparable to that of training set throughout the training which confirms model didn't over fit for the training data.

6.1.3 Evaluation

After training for 8 epochs, the model is evaluated on the test data. The BLEU and ROUGE evaluation metrics are used to evaluate the model.

BLEU Score
The average BLEU score for n-grams 1, 2, 5 and 10 is as follows:

- 1-grams BLEU Score: 0.9544114683715649
- 2-grams BLEU Score: 0.9378854888760932
- 5-grams BLEU Score: 0.8867411989388778
- 10-grams BLEU Score: 0.7982317463855622

The above observation shows that good performance was achieved by the model on the given testing data. The average BLEU-1 score of 0.95 indicates that the generated DSL closely matches the reference DSL, suggesting that the model is effective at predicting the structure of short sequences. Although the BLEU score decreases as the n-gram size increases, a good performance is still maintained, indicating that the model excels at predicting short sequences. The BLEU-10 score of 0.79 further demonstrates that longer sequences were predicted accurately, showing the model's ability to handle more complex and extended sequences.The individual BLEU-10 scores, sorted in order, were plotted in the graph. It can be observed that the BLEU-10 score is higher than 0.6 for most of the samples, reflecting consistent and reliable performance across a range of test cases. Additionally, numerous samples have BLEU scores above 0.9, supporting the conclusion that the model performed well across the testing data. This indicates that the model consistently generated high-quality results for both short and long sequences, confirming its reliability and effectiveness for the task.

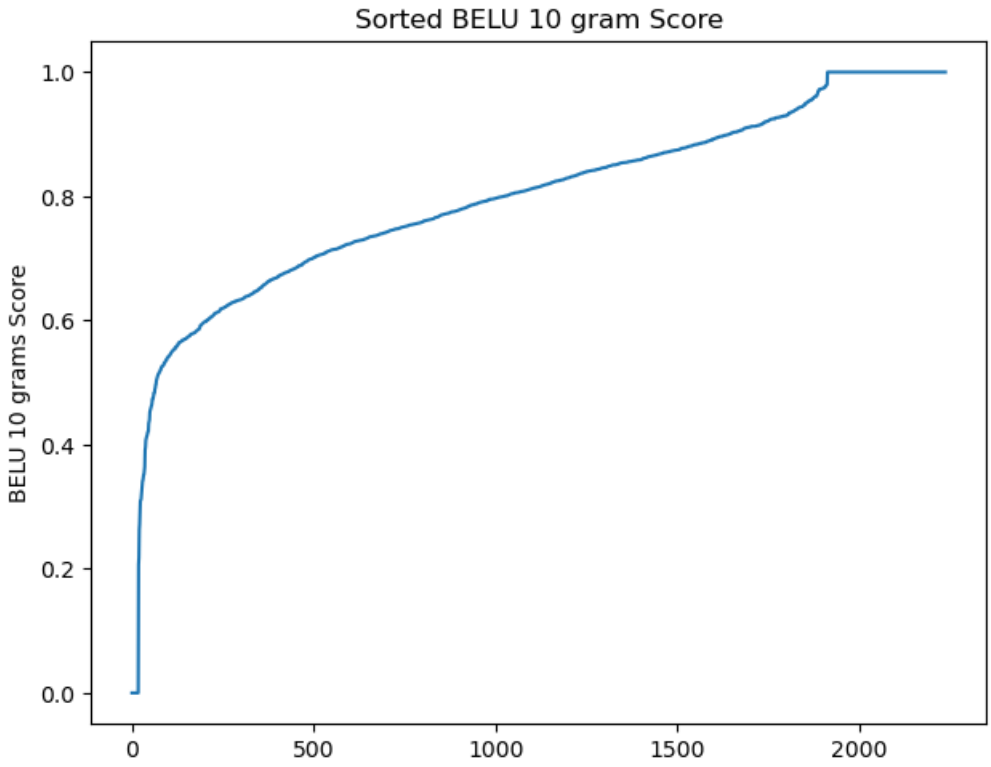

Figure 6-3: BLEU-10 Scores in sorted order

ROUGE Score

The average ROUGE Scores are tabulated below: Inserting table:

Table 6-1: ROUGE Scores for test data

Metric	Precision	Recall	F1-Score
ROUGE-1	0.9481	0.9486	0.9652
ROUGE-5	0.7806	0.8112	0.7948
ROUGE-L	0.9428	0.9791	0.9598

From above table, ROUGE-1 has high F1-score of 0.96. This indicates the predicted tokens matches 96% with the reference tokens. The ROUGE-5 F1-score is 0.79. It has decreased drastically but it is still a good score. The high recall of 0.97 for ROUGE-L indicates that the generated DSL captures most of the longest common subsequences from the reference. This indicates the generator maintains correct sequence and structure of DSL. The graph below shows that the F1-score for almost all samples is above 0.9 which is good performance.

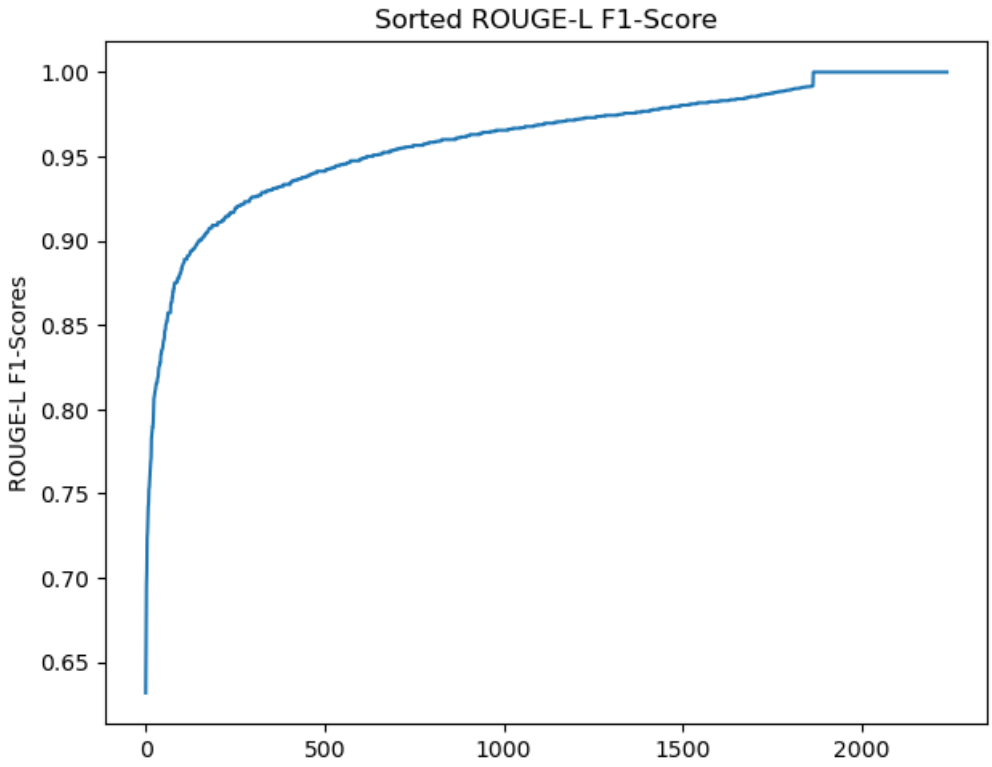

Figure 6-4: ROUGE-L F1-Scores in sorted order

6.2 Dataset 2

By improving the dataset generation process the new sets of dataset comprising of about 19000 images is generated. The new samples of elements are collected and data is generated. 13510 images were used for training and 5790 images were used for validation purpose. The three different models were trained for the given data. The 100 hand drawn images were used to evaluate the performance of all three models. The associated dsl code for the hand drawn images were provided manually by human evaluation based on the rule.

6.2.1 Model 1(Compact Convolutional Transformer Encoder with Transformer Decoder)

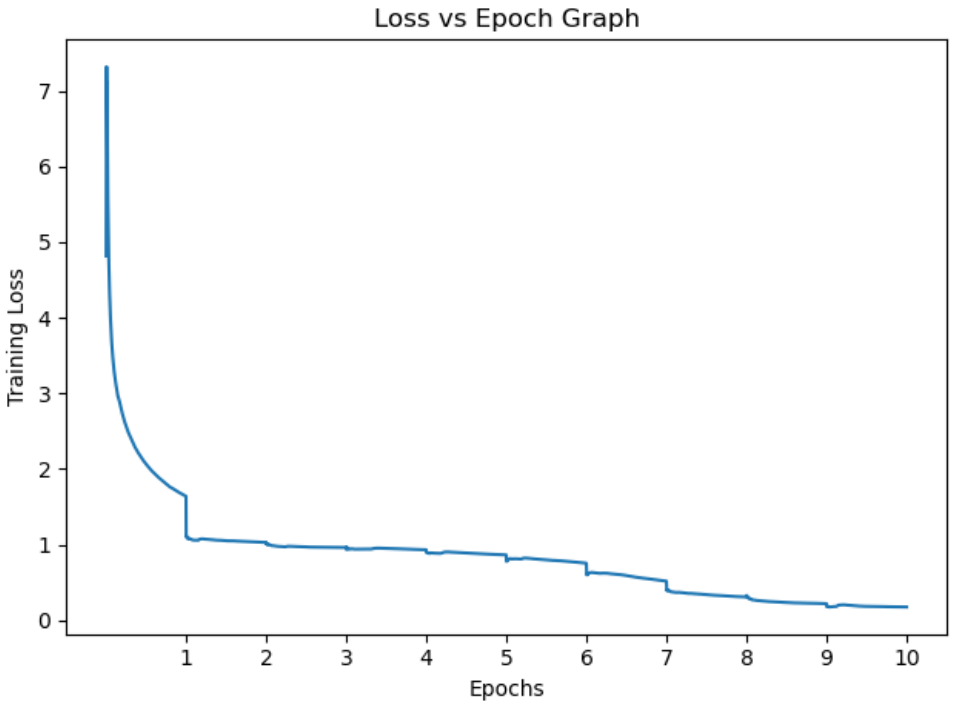

Figure 6-5: Training loss vs Epoch graph(Model 1)

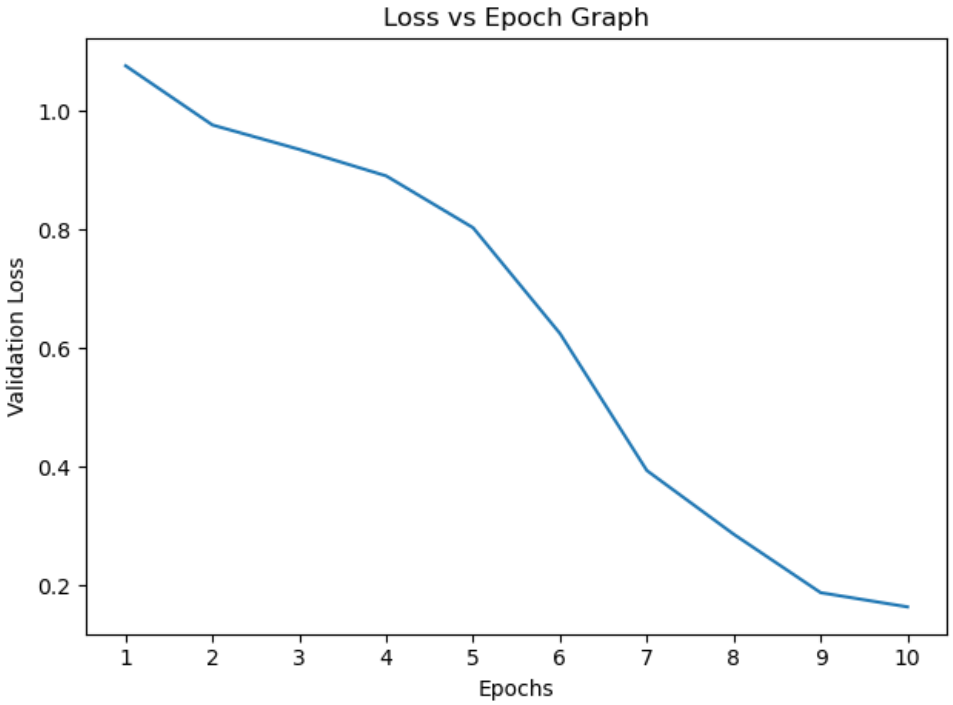

Figure 6-6: Validation loss vs Epoch graph(Model 1)

Initially, the loss for the model is very high. The model get familiar with the nature of the data and associated output in the first epoch and the loss decreased rapidly in the first epoch. During next 9 epochs, the loss gradually decreased from 1 to near 0. The decreasing nature of the graph indicates the model is learning the nature of data quiet well. The loss has gradually decreased in both training and validation. The loss decreased from 4.81 to 0.17 at the end of 10th epoch. The masked accuracy of 0.93 was achieved for training data and same for the validation data.

Evaluation

The ROUGE and BlEU metric is used to evaluate the performance of the model in 100 hand drawn images.

BLEU Score

The average BLEU score for n-grams 1,2,5 and 10 is follows:

- 1-grams BLEU Score: 0.7471271688303959
- 2-grams BLEU Score: 0.6580029789136941
- 5-grams BLEU Score: 0.42442962967602715
- 10-grams BLEU Score: 0.30114337436735683

ROUGE Score

The average rouge score obtained for the test data are tabulated below.

Table 6-2: ROUGE Scores for test data (model1)

Metric	Precision	Recall	F1-Score
ROUGE-1	0.7124	0.7768	0.7385
ROUGE-5	0.4218	0.4732	0.4467
ROUGE-L	0.6261	0.6823	0.6487

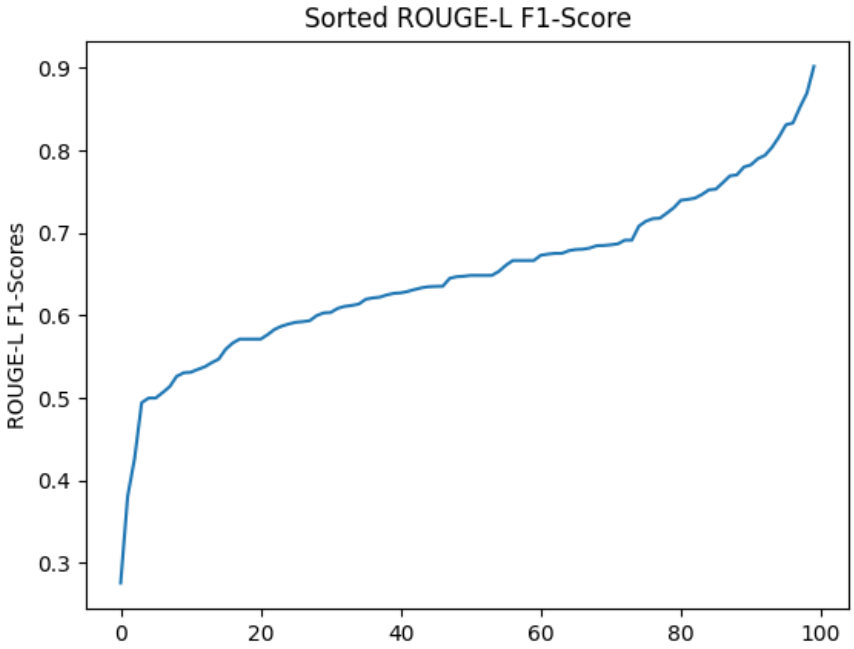

Figure 6-7: Sorted Rouge-L F1-Score(Model 1)

It is seen that ROUGE-L score is more than 0.5 for most of data leaving 3 images. The most of the data have score between 0.5 and 0.7. There are also many number of images with score above 0.7. The highest score is 0.9019 and the lowest score is 0.2758. The best and worst case test data is shown below:

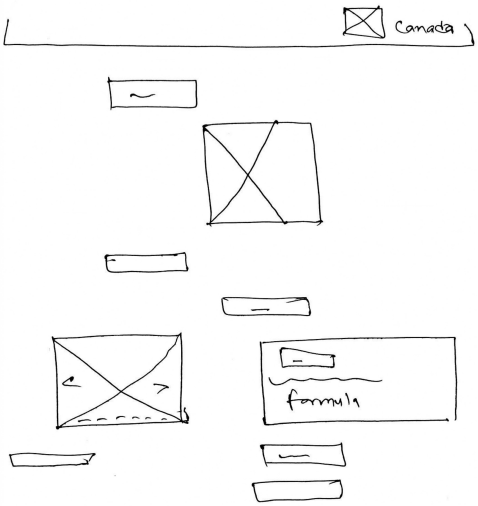

Figure 6-8: Input Image(ROUGE-L score 0.901)

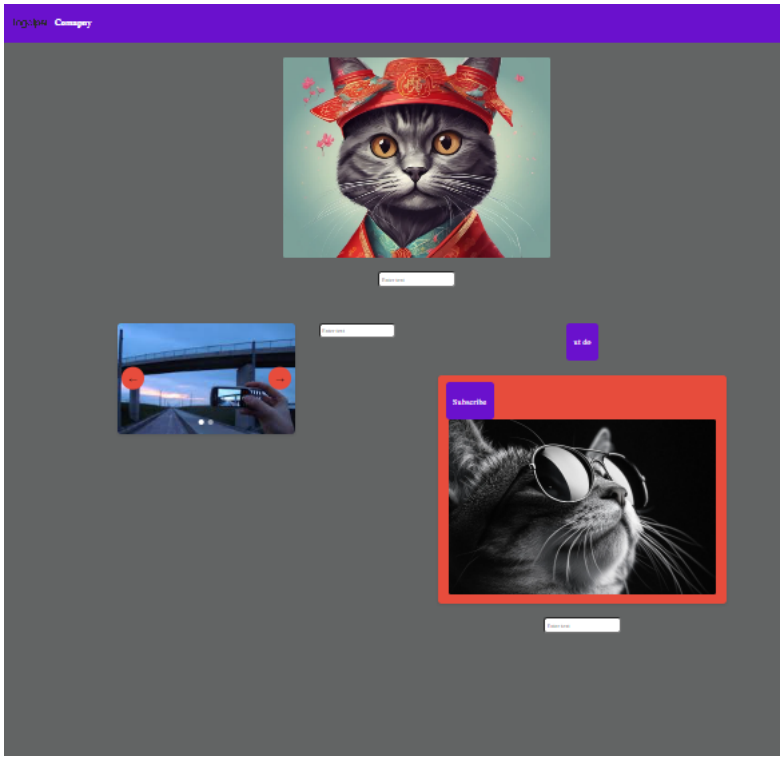

Figure 6-9: Output(ROUGE-L Score 0.901)

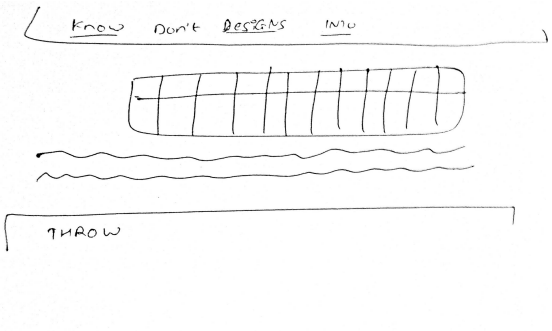

Figure 6-10: Input Image(ROUGE-L Score 0.2758)

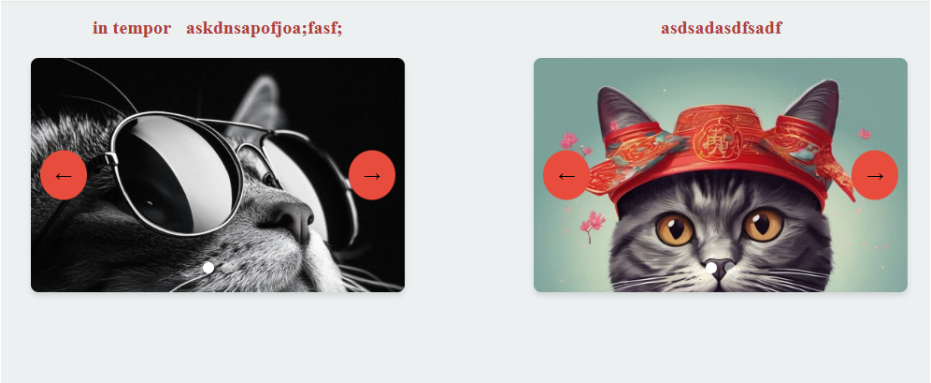

Figure 6-11: Output(ROUGE-L score 0.2758)

6.2.2 Model 2(Vision Transformer Encoder with Transformer Decoder)

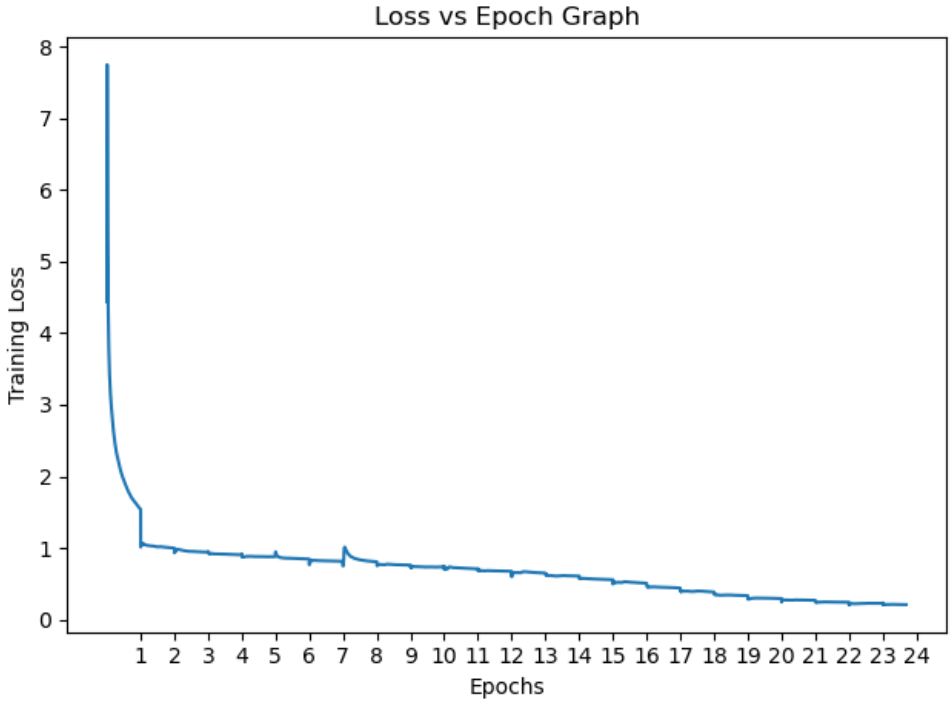

Figure 6-12: Training Loss Vs Epoch(Model 2)

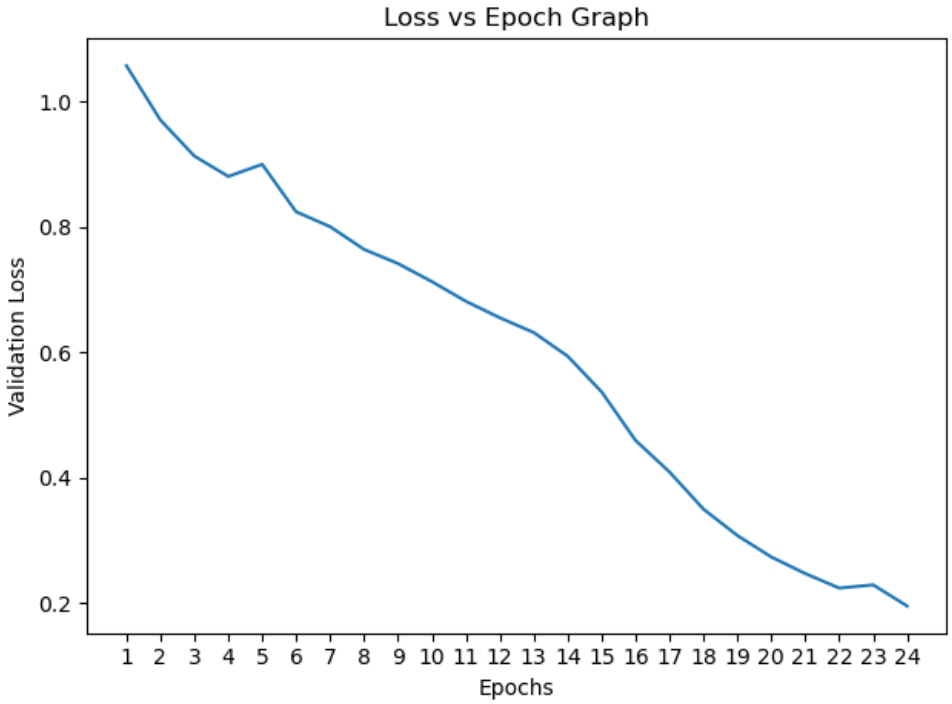

Figure 6-13: Validation Loss vs Epoch(Model 2)

The patch of size 16x16 is extracted from the image of size (848,608). The loss of the training data decreased rapidly from 7.23 to 1.54 in the first epoch of training. It indi-

cates the model was able to learn about the data during the training. In the subsequent iterations, the loss of both training and validation data decreased gradually. The rate at which the loss decreased is found to be slower than for model 1. The training loss of 0.2 was reached after training for 24 epochs. The training was stopped as the loss start to increase. The masked accuracy of 0.91 was reached at the end of the training.

Evaluation

The ROUGE and BlEU metric is used to evaluate the performance of the model in 100 hand drawn images.

BLEU Score

The average BLEU score for n-grams 1,2,5 and 10 is follows:

- 1-grams BLEU Score: 0.6798317754323199
- 2-grams BLEU Score: 0.5806978773972931
- 5-grams BLEU Score: 0.3387053560764535
- 10-grams BLEU Score: 0.25798579488789025

ROUGE Score

The average rouge score obtained for the test data are tabulated below.

Table 6-3: ROUGE Scores for test data(Model 2)

Metric	Precision	Recall	F1-Score
ROUGE-1	0.6329	0.7560	0.6749
ROUGE-5	0.3842	0.4589	0.4091
ROUGE-L	0.5244	0.6256	0.5583

From above table, ROUGE-1 recall is 0.756 that indicates that 75% predicted tokens matched reference tokens of the data. While considering 5 tokens at a time, the score reached 0.45. For the longest subsequence, the recall score obtained is 62%. The graph consisting of ROUGE L scores in the sorted order is presented below:

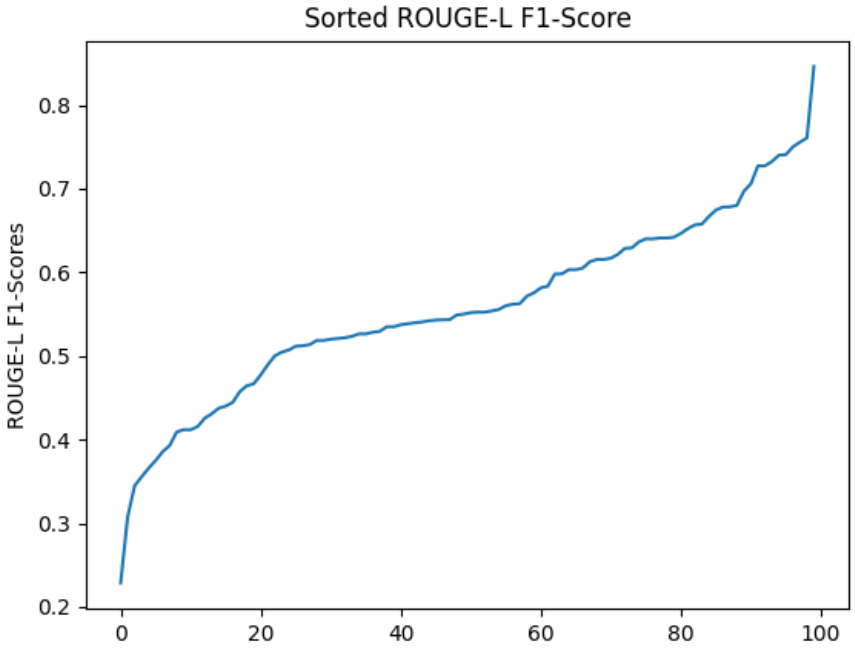

Figure 6-14: Sorted ROUGE-L F1-Scores in sorted order(Model 2)

It is seen that ROUGE-L score is more than 0.5 for most of data. The most of the data have score between 0.5 and 0.7. The few samples have the score above the 0.7. The minimum score is 0.22 and the maximum score obtained is 0.845. The best and worst case test data is shown below:

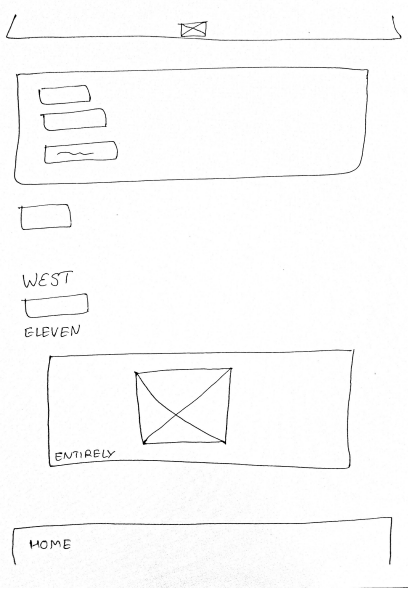

Figure 6-15: Input Image(ROUGE-L score 0.845)

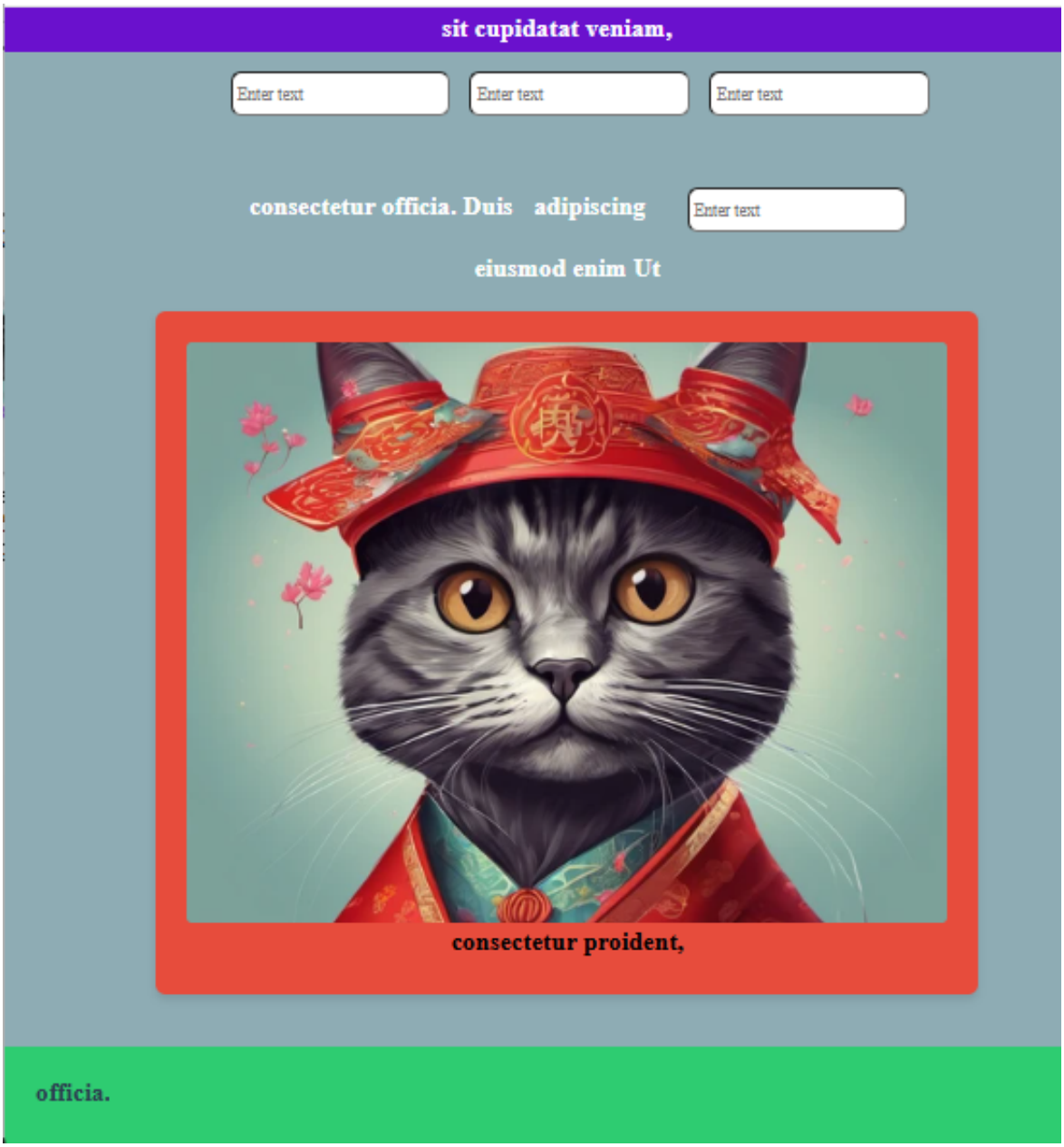

Figure 6-16: Output(ROUGE-L score 0.845)

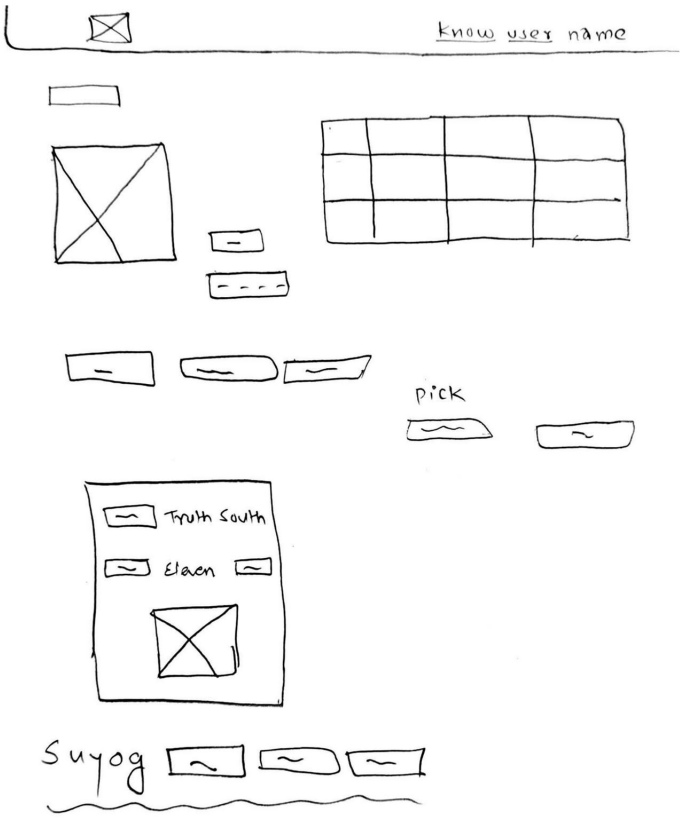

Figure 6-17: Input Image(ROUGE-L score 0.22)

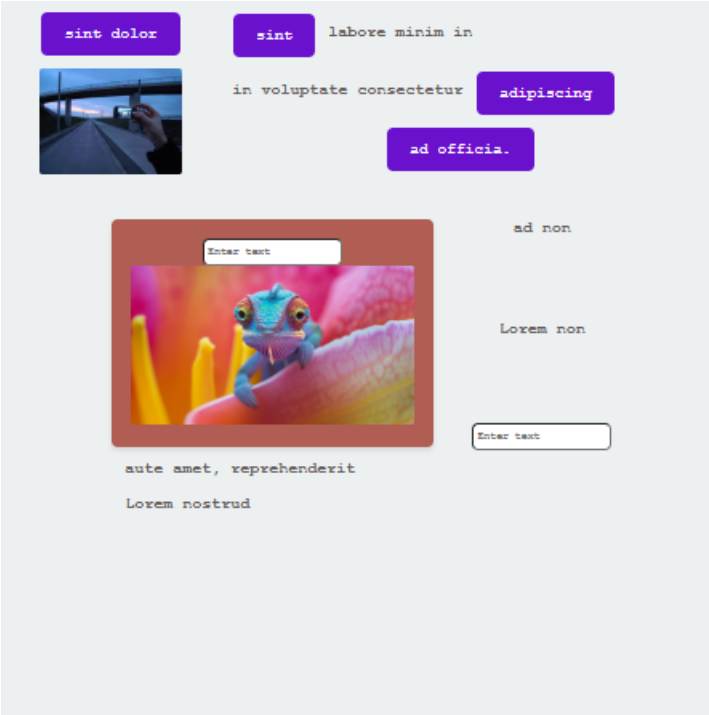

Figure 6-18: Output(ROUGE-L score 0.22)

6.2.3 Model 3(Convolutional Encoder with Transformer Decoder)

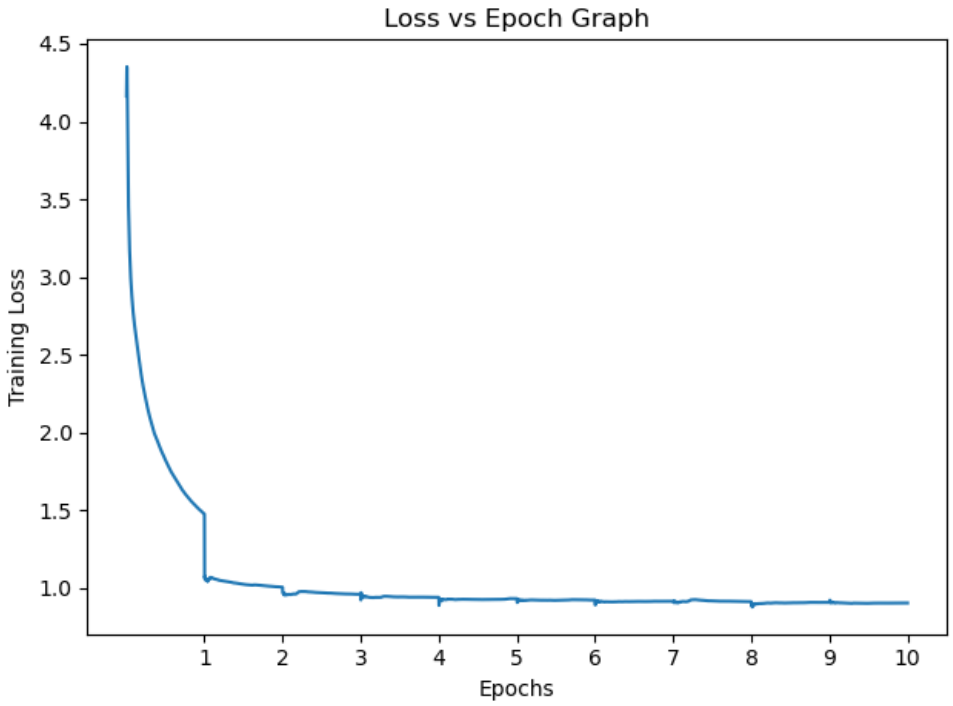

Figure 6-19: Training Loss vs Epoch(model 3)

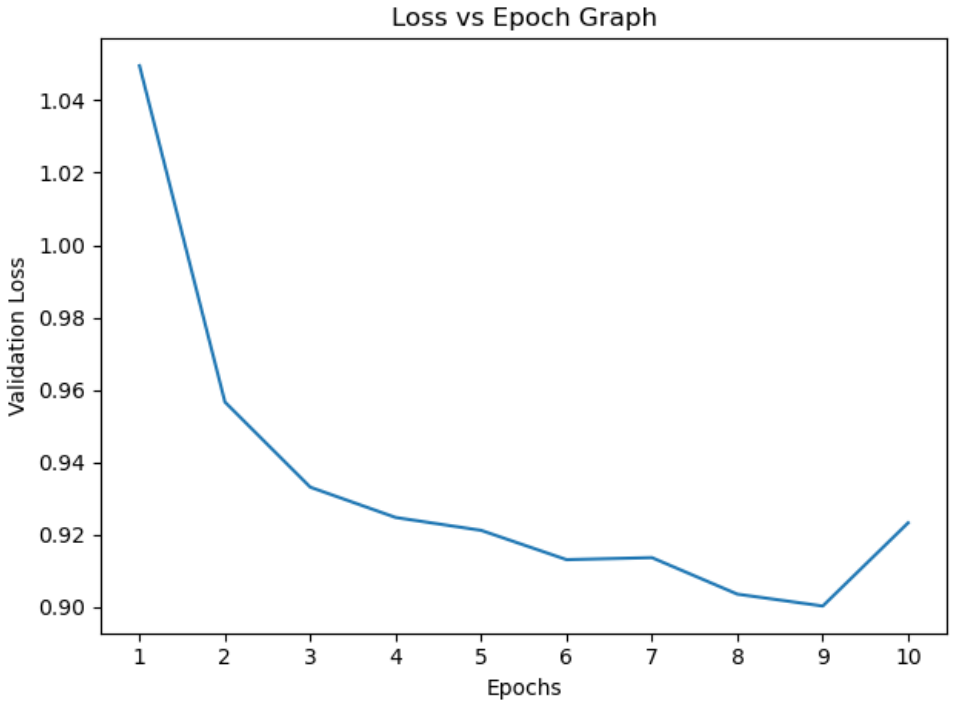

Figure 6-20: Validation Loss vs Epoch(model 3)

Like other models, the loss of the model was high at the beginning. The loss decreased slowly from 4.2. The training loss doesn't seem to decrease after 4th epoch. The loss of

the training data is in range of about 0.9 after 2nd epoch. The training was stopped after the loss start to increase. The training loss of 0.93 was encountered at the end of 10th epoch. The masked accuracy reached the value of 0.63 for the training data. As the loss of the model didn't decreased to suitable value the model was not further evalutated.

6.2.4 Comparison of Model 1 and 2

Both the model consists of transformer encoder and decoder. In model 2, the patches were extracted from the images were input to the transformer. In model 1, the feature map extracted from the convolutional tokenizer is flattened and provided to the model. The training of the model 2 seems to be slower than the model 1. The model 1 uses learnable positional encoding while model 2 uses sinosudoal positional encoding. The model 1 achieved about 90% accuracy after 10 epochs while model 2 took more than 20 epochs.By comparing the ROUGE and BLEU score obtained for both models, model 1 outperforms model 2 in terms of metrics. Model 1(4675007) has more trainable params than the model 2 (3210527).

6.3 Customization

User can style and customize the layout and visuals of the generated webpage.

6.3.1 UI

Here, is the screenshot of the interface which can be used to change the style of the generated web page. First, they have to upload the image and then, they can customize the webpage. Here, user can save the desired webpage and they will get html, CSS for that page.

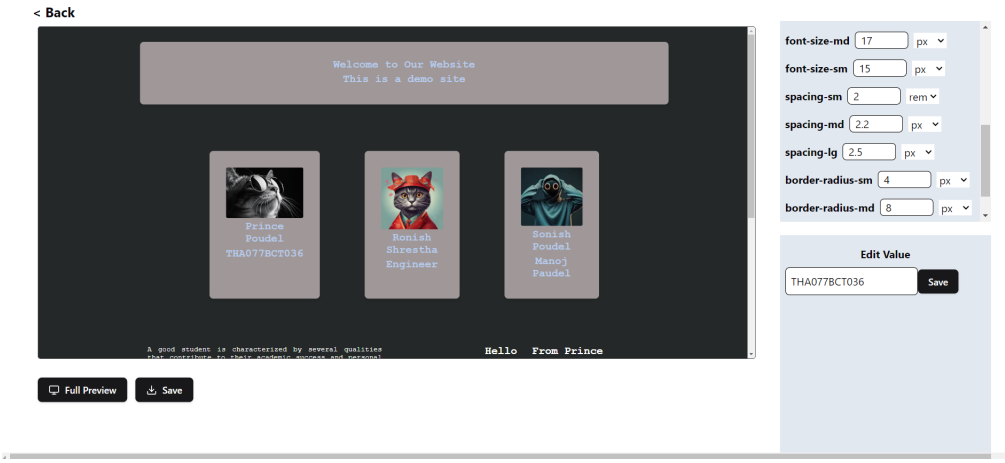

Figure 6-21: User Interface

6.3.2 Full Webpage

This is the webpage that a user can get after customization.

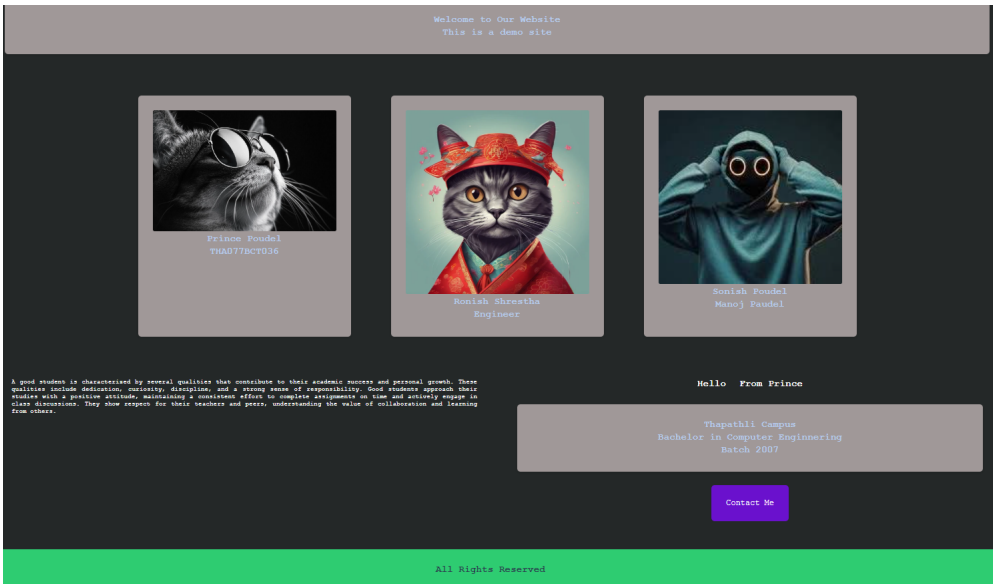

Figure 6-22: Full webpage

6.3.3 Customization Options

Different customization options are available to change the style of webpage. Mainly, user can change the color, text, size and text font. All the options available are given below.

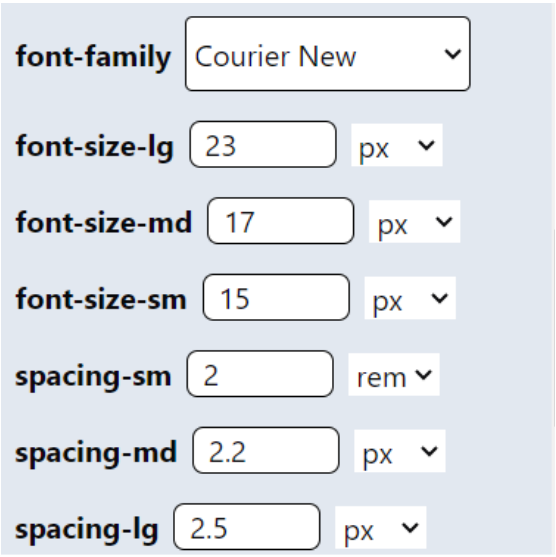

Figure 6-23: Font Customization

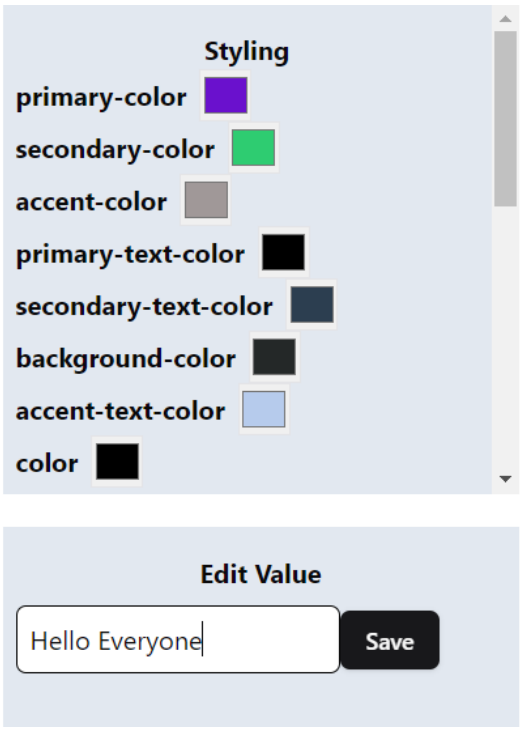

Figure 6-24: Color Customization

6.3.4 JSON file

This is the sample of the JSON file which have information about the element and the styles. It also carries the changes specified by the user through the User interface. Node key helps in carrying the information about the hierarchical structure of the webpage.

```
{
    "node": {
      "element": "root",
      "nodes": [
        {
          "element": "header",
          "nodes": [
            {
              "element": "flex",
              "nodes": [
                {
                  "element": "logodiv",
                  "nodes": [
                    {
                      "element": "image",
```

```
            "nodes": [],
          }
        ]
      }
    ]
  }
},
{
  "element": "container",
  "nodes": [
    {
      "element": "row",
      "nodes": [
        {
          "element": "div-6",
          "nodes": [
            {
              "element": "image",
              "nodes": [],
              "url": "https://..........."
            }
          ]
        }
      ]
    },
    {
      "element": "row",
      "nodes": [
        {
          "element": "div-12",
          "nodes": [
            {
              "element": "text",
              "nodes": [],
              "text": "About Us"
            },
            {
```

```
                    "element": "paragraph",
                    "nodes": [],
                    "text": "We ............"
                  }
                ]
              }
            ]
          }
        ]
      }
    ],
    "styles": {
      "accent-color": "#e74c3c",
      "accent-text-color": "black",
      "background-color": "#ecf0f1",
      "border-radius-lg": "12px",
      "border-radius-md": "8px",
      "border-radius-sm": "4px",
      "color": "#1c1717",
      "font-family": "Arial",
      "font-size-lg": "25px",
      "font-size-md": "16px",
      "font-size-sm": "18px",
      "primary-color": "#6a11cd",
      "primary-text-color": "white",
      "secondary-color": "#2ecc71",
      "secondary-text-color": "#2c3e50",
      "spacing-lg": "2rem",
      "spacing-md": "1.5rem",
      "spacing-sm": "1rem"
    }
  }}
```

6.3.5 Html File

This is the sample of the HTML and CSS file which is compiled from the JSON.

```
<head>
```

```
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Generated Page</title>
</head>
<body>
  <div id="root" class="root" data-id="110">
    <header class="header" data-id="111">
      <div class="flex" data-id="112">
        <div class="logodiv" data-id="113">
          <img
            src="https://..."
            class="image"
            data-id="114"
          />
        </div>
      </div>
    </header>

    <div class="container" data-id="115">
      <div class="row" data-id="116">
        <div class="div-6" data-id="117">
          <img
            src="https://....
            class="image"
            data-id="118"
          />
        </div>
      </div>
      <div class="row" data-id="119">
        <div class="div-12" data-id="120">
          <div class="text" data-id="121">About Us</div>
          <p class="paragraph" data-id="122">We are</p>
        </div>
      </div>
    </div>
  </div>
</body>
```

```
</html>
```

6.4 Sample Generated Datasets

Sample 1

```
container {
    row {
        div-6 {
            image
            paragraph
        }
        div-6 {
            flex-r {
                button
                button
                text
            }
            carousel
            button
        }
    }
    row {
        div-9 {
            carousel
            input
            text
        }
        div-3 {
            card {
                image
                paragraph
                text-c
            }
            paragraph
        }
    }
    row {
```

```
            div-3 {
                card {
                    input
                    input
                }
                input
            }
            div-3 {
                card {
                    paragraph
                    text-c
                }
                button-c
            }
            div-6 {
                paragraph
                button
                text
            }
        }
    }
    footer {
        text-c
    }
```

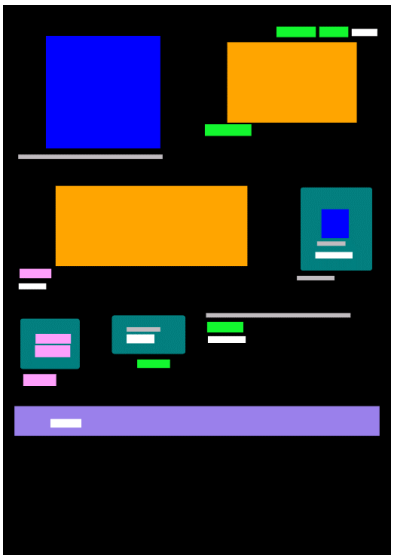

Figure 6-25: Rendered HTML

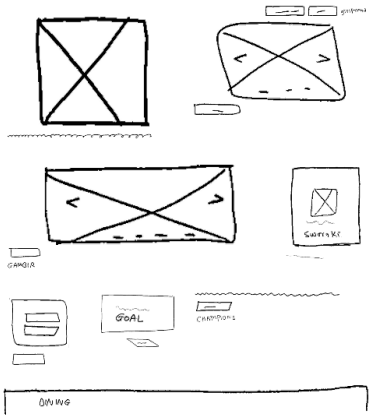

Figure 6-26: Sketch

Sample 2

```
header {
    flex-sb {
        logodiv {
            image
            text
        }
        nav {
            navlink
            navlink
            navlink
            navlink
        }
    }
}
container {
    row {
        div-9 {
            input
            flex-sb {
                button
                button
                button
                text
            }
            table
            button
        }
        div-3 {
            card {
                paragraph
                text-c
                paragraph
            }
            paragraph
        }
    }
```

```
    row {
        div-3 {
            button-c
            text
            image
            input
        }
        div-6 {
            paragraph
            carousel
            paragraph
        }
        div-3 {
            text
            button
            image
        }
    }
    row {
        div-3 {
            text-c
            card {
                button
                image
            }
            input
        }
    }
}
```

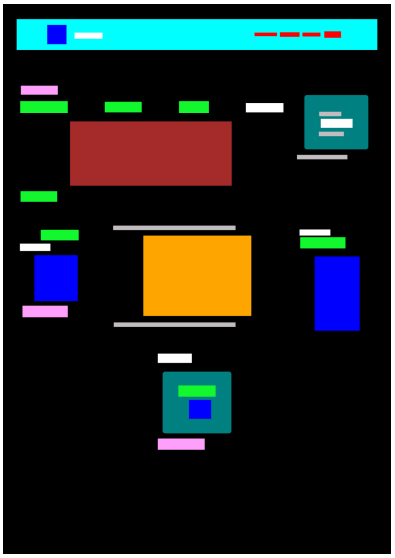

Figure 6-27: Rendered HTML

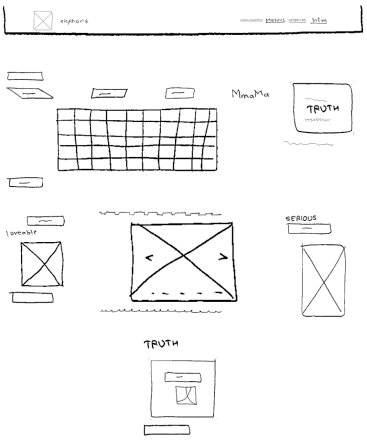

Figure 6-28: Sketch

7 FUTURE ENHANCEMENT:

- We can Integrate JavaScript to create interactive elements and dynamic behaviors within the generated user interfaces.
- The system can be extended to support multiple frontend frameworks including React, Angular, and Vue.js to accommodate different development environments.
- We can implement responsive design capabilities where generated interface automatically adapts to different screen sizes and devices.
- We can increase the elements pool to include more complex UI components such as calendar, navigation menus etc.

8 CONCLUSION

Our project has successfully developed an artificial Intelligence model that transforms hand-drawn sketches into functional HTML/CSS code. By using transformer models, particularly the Compact Convolutional Transformer Encoder which achieved an impressive BLEU score of 0.901. We've created a bridge between design ideas and working prototypes. Convolution layer used in our Encoder helped to extract the important features which reduce the size of the input and making it easier to train a model. We built a custom Domain Specific Language to represent UI elements and their hierarchical structure, and then created a user-friendly interface that lets users to easily customize text, images, and fonts. The dataset generator which we created solved the training data challenge by automatically producing required sketch and its associated DSL code. This tool helps designers and developers work faster by turning layout drawn on paper into usable code in seconds which saves time for more creative tasks. While our current version focuses only on static layouts, the foundation is in place for future enhancements like adding JavaScript interactivity, supporting popular frameworks, and expanding the available UI elements. Overall, our project demonstrates how AI can be used to create a functional code from initial design.

9 APPENDICES

Appendix A: Project Schedule

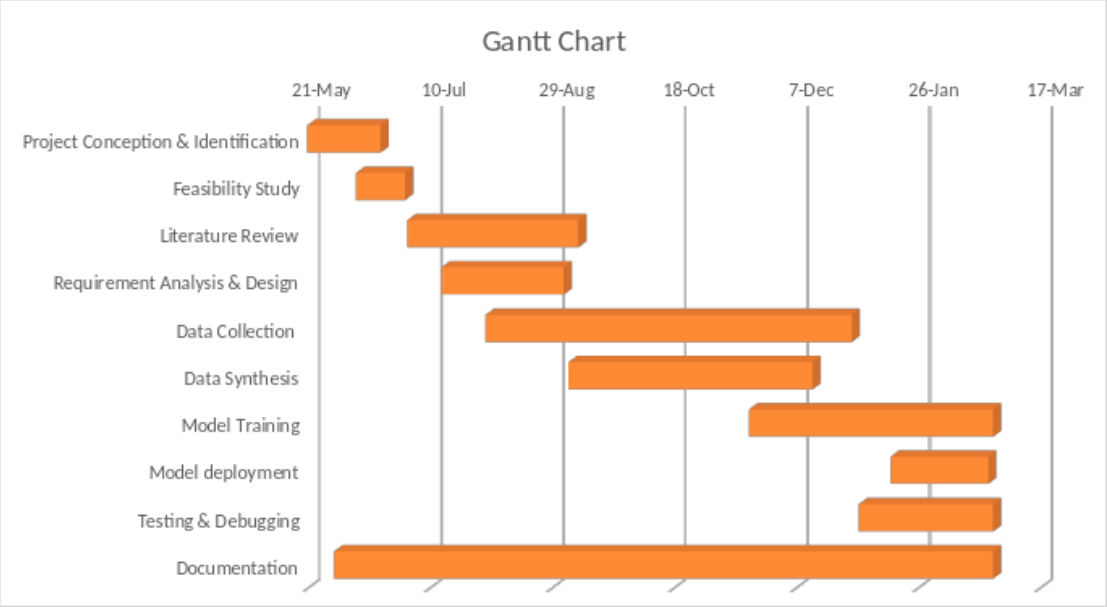

Figure 9-1: Gantt Chart

Appendix B: DSL Generation Rules

```
1. Dictionary for Child Elements
graph={
    'root':['header','container','footer'],
    'header':['flex'],
    'nav':['navlink'],
    'logodiv':['image','text'],
    'container':['row'],
    'row':['div-3','div-6','div-9','div-12'],
    'div-3':['text','paragraph','image','card','input','button',],
    'div-6':['text','paragraph','image','card','carousel','input'
    ,'button','flex'],
    'div-9':['text','paragraph','image','card','carousel','input',
    'table','button','flex'],
    'div-12':['text','paragraph','image','card','carousel','input',
    'table','button','flex'],
    'flex':['text','button'],
    'card':['text','paragraph','image','input','button','flex'],
    'footer':['text']
}
```

```
2. Rules for Child Elements
rules = {
    'root': {'inOrder': True},
    'logodiv': {'inOrder': True},
    'header':{'min':1,'max':1},
    'container': {'min': 1, 'max':3},
    'row': {'combinations':True,'0': divCombinations,
    '1':divCombinations2,'proba':0.9},
    'div-3': {'min': 2, 'max': 4},
    'div-6': {'min': 2, 'max': 4},
    'div-9': {'min': 2, 'max': 4},
    'div-12': {'min': 2, 'max':4},
    'card': {'min': 2, 'max': 3},
    'footer': {'min': 1, 'max': 1},
    'nav':{'min': 1, 'max': 5},
    'flex':{'min':2,'max':4},
}

3. DSL to HTML mappings = {
    "opening-tag": "{",
    "closing-tag": "}",
    "body": "\r\n {} <style>\nmax-width: 900px;
    \nmax-height: 300px !important;\n</style>\n",
    "root": "\r\n<div class=\"root\">{}</div>\r\n",
    "header": "\r\n<header class=\"header\">{}</header>\r\n",
    "nav": "\r\n<nav class=\"nav\">{}</nav>\r\n",
    "navlink": "\r\n<a href=\"#\" class=\"navlink\">[]</a>\r\n",
    "logodiv": "\r\n<div class=\"logodiv\">{}</div>\r\n",
    "container": "\r\n<div class=\"container\">{}</div>\r\n",
    "row": "\r\n<div class=\"row\">{}</div>\r\n",
    "div-3": "\r\n<div class=\"div-3\">{}</div>\r\n",
    "div-6": "\r\n<div class=\"div-6\">{}</div>\r\n",
    "div-9": "\r\n<div class=\"div-9\">{}</div>\r\n",
    "div-12": "\r\n<div class=\"div-12\">{}</div>\r\n",
    "flex": "\r\n<div class=\"flex\">{}</div>\r\n",
    "flex-sb": "\r\n<div class=\"flex-sb\">{}</div>\r\n",
    "flex-c": "\r\n<div class=\"flex-c\">{}</div>\r\n",
    "flex-r": "\r\n<div class=\"flex-r\">{}</div>\r\n",
```

```
    "text": "\r\n<div class=\"text\">{}</div>\r\n",
    "text-c": "\r\n<div class=\"text-c\">{}</div>\r\n",
    "text-r": "\r\n<div class=\"text-r\">{}</div>\r\n",
    "paragraph": "\r\n<p class=\"paragraph\">{}</p>\r\n",
    "image": "\r\n<img src=\"placeholder.jpg\" class=
    \"image\">\r\n",
    "card": "\r\n<div class=\"card\">{}</div>\r\n",
    "input": "\r\n<input type=\"text\" class=\"input\"
     placeholder=\"Enter text\">\r\n",
    "button": "\r\n<button class=\"button\">{}</button>\r\n",
    "button-c": "\r\n<button class=\"button-c\">{}</button>\r\n",
    "button-r": "\r\n<button class=\"button-r\">{}</button>\r\n",
    "footer": "\r\n<footer class=\"footer\">{}</footer>\r\n",
    "table": "\r\n<table class=\"table\">{}</table>\r\n",
    "carousel": "\r\n<div class=\"carousel\">{}</div>\r\n"
}

4. Color Mapping for each elements = {
        'paragraph': np.array([193, 188, 192]),
        'text': np.array([255, 255, 255]),
        'button': np.array([19, 247, 47]),
        'navlink': np.array([255, 0, 0]),
        'carousel': np.array([255, 165, 0]),
        'table': np.array([165, 42, 42]),
        'input': np.array([255, 159, 252]),
        'image': np.array([0, 0, 255]),
        'header': np.array([0, 255, 255]),
        'footer': np.array([154, 128, 235]),
        'card': np.array([0, 128, 128]),
    }
```

References

[1] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," 2017. [Online]. Available: https://arxiv.org/abs/1705.07962

[2] S. Suleri, V. P. Sermuga Pandian, S. Shishkovets, and M. Jarke, "Eve: A sketch-based software prototyping workbench," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3290607.3312994

[3] B. B. Adefris, A. B. Habtie, and Y. G. Taye, "Automatic code generation from low fidelity graphical user interface sketches using deep learning," in *2022 International Conference on Information and Communication Technology for Development for Africa (ICT4DA)*, 2022, pp. 1–6.

[4] D. Baule, C. Gresse von Wangenheim, A. von Wangenheim, J. C. R. Hauck, and E. C. Vargas Júnior, "Automatic code generation from sketches of mobile applications in end-user development using deep learning," *arXiv preprint arXiv:2103.05704*, 2021. [Online]. Available: https://arxiv.org/abs/2103.05704

[5] J. Doe and J. Smith, "Sketch2fullstack: Generating skeleton code of full stack website and application from sketch using deep learning and computer vision," *arXiv preprint arXiv:2211.14607*, 2022.

[6] M. Taylor and S. Lee, "Skcoder: A sketch-based approach for automatic code generation," *arXiv preprint arXiv:2302.06144*, 2023.

[7] S. R, S. P. K. R, V. R. S, V. W, and L. Srinivasan, "Stc (sketch to code)-an enhanced html & css autocode generator from handwritten text and image using deep learning," in *2024 2nd International Conference on Networking and Communications (ICNWC)*, 2024, pp. 1–6.

[8] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021. [Online]. Available: https://arxiv.org/abs/2010.11929