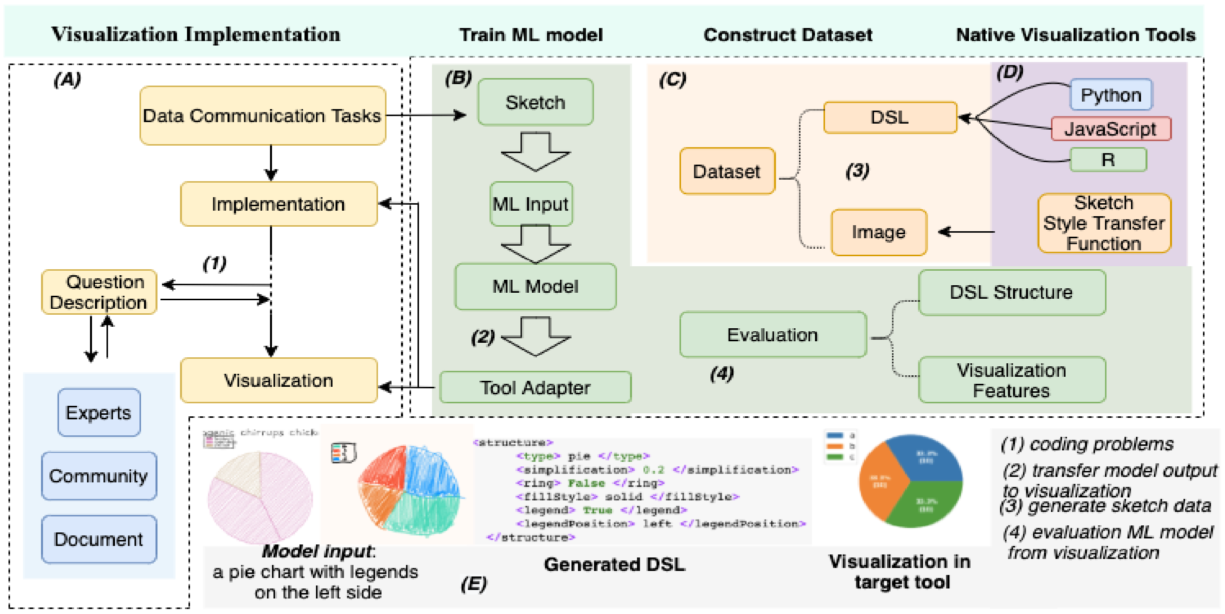
# Analyzing the Feasibility of Generating Data Visualizations from Hand-drawn Sketches Using Deep Learning



```
<structure>
        <type> pie </type>
        <simplification> 0.2 </simplification>
        <ring> False </ring>
        <fillStyle> solid </fillStyle>
        <legend> True </legend>
        <legendPosition> left </legendPosition>
  </structure>
```

Fig. 1. An Overview of the Sketch2Vis problem. Block(A) shows the Sketch2Vis problem where a hand-drawn sketch of a desired visualization is provided to a deep learning model to generate visualizaton code. Our work covers three areas of this challenge: dataset construction, machine learning (ML) model improvement, and dataset scaling. Block(B) shows the high-level approach architecture. Block(C) and Block(D) show how we use Domains-specific Languages (DSLs) and style transfer to aid in dataset generation. Block(E) shows an example (Input, intermediate output, and visualization output) of the Sketch2Vis model.

**Abstract**—Data visualization has become a vital tool to help people understand the driving forces behind real-world phenomena. Although the learning curves of visualization tools have been reduced significantly, domain experts still may require training to use them. This paper investigates the feasibility of using deep learning techniques and tools to generate the source code for multi-platform data visualizations automatically from hand-drawn sketches provided by a domain expert. The idea is similar to how an expert might sketch on a cocktail napkin and ask a software engineer to implement the sketched visualization. We explore key challenges in generating visualization code from hand-drawn sketches, particularly model training challenges, since acquiring a large dataset of sketches paired with visualization source code is expensive. We present solutions for these problems and conduct experiments that demonstrate the feasibility of generating visualizations from hand-drawn sketches. The best models tested reach an structural accuracy of 95% in generating correct data visualization source code from hand-drawn sketches of visualizations.

**Index Terms**—Data Visualization, Deep Learning

## 1 Introduction

Data visualization is a vital tool that enables people to better understand the driving forces behind real-world phenomena [13]. These visualizations help provide insights by creating graphical representations of data element relationships, trends, and dominant features. Many visualization tools are available, ranging from programming-based visualization libraries (such as Matplotlib [42] and D3 [4]) to user-friendly graphical user interface (GUI) apps (such as Tableau [33]) for building visualizations interactively.

Building good data visualizations from raw data is hard. Programmers need training to use data visualization tools competently, including tools that use GUI-based interfaces [10]. A consequence of this need for training is that domain experts who need visualizations to understand physical systems (e.g., power grids, pedestrian traffic, or healthcare processes) often team up with a visualization drafter (e.g., a software engineer or data scientist) to build a data visualization collaboratively [38].

For example, consider a scenario where a physician unfamiliar with data visualization tools needs several visualizations produced to understand how data (e.g., heart rate, pulse oxygen, blood glucose data) from medical devices attached to a patient correlate with changes in the patient's blood pressure. The physician could state these requirements to the drafter and describe the format they want the medical device data presented in, which could then be used to generate the visualizations. The physician might also draw a sketch to describe roughly what they expect the final visualization to look like.

In this scenario, the sketch would provide an abstract representation

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

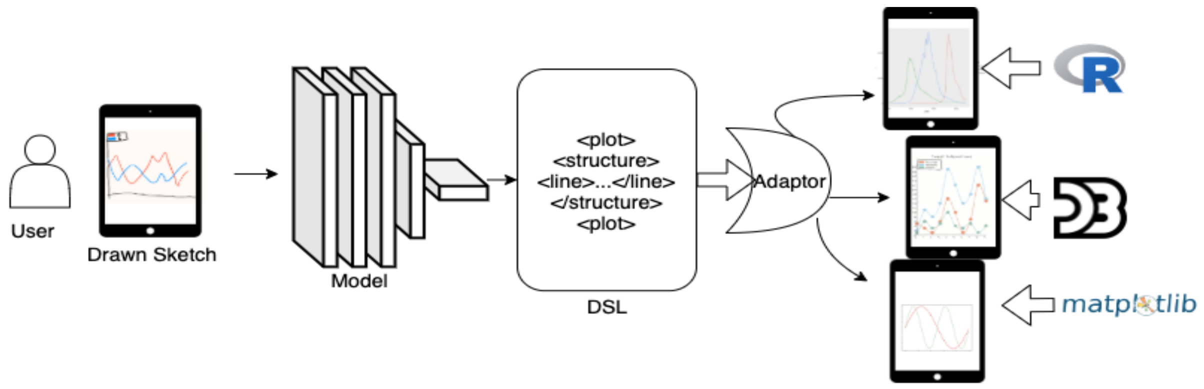


Fig. 2. A Scenario of the Sketch2Vis Problem in Mobile Devices.

or set of constraints that the final visualization of the data is expected to adhere to (e.g., a drawing of a line plot with separate series for each of heart rate, pulse oxygen, blood glucose vs. blood pressure). The drafter would take the high-level sketch of the requirements describing the type of visualization and a description of the data to use (e.g., provided by the labels of the sketch). They could then instantiate a concrete visualization in source code and connect it to the appropriate data corresponding to labels in the sketch.

**Research question: Can deep learning be used to generate visualizations from hand-drawn sketches automatically?** Drawing is a natural way for domain experts to express their visualization goals since hand-drawn sketches are rich in information without specifying visualization terminologies. By leveraging sketches as a type of input for data visualization, visualization tools could potentially make it easier for domain experts and untrained professionals to explore data sets. Moreover, as a natural way to interact with mobile devices, sketches can be integrated into mobile data exploration tools [32] to help users operate visualization features easier on mobile devices.

We call the problem of translating human sketches into data visualization source code the "Sketch2Vis problem." Figure 2 shows a scenario of applying the Sketch2Vis problem on mobile apps. As shown in this figure, users can draw sketches on a mobile device without specifying visualization implementation details and a deep learning model can then generate source code for the desired visualization in multiple target visualization libraries. We focus on the generation of the source code to realize the visualization-only at this point.

The generation of complex queries from natural language to select data for the plot is a separate and rich domain of research [35,40,41,45]. Combining these approaches could be performed to fully automate the approach described in this paper. This paper, however, assumes a manual data selection step from a drop-down menu.

This paper presents the results of our research on generating multi-platform visualization code from hand-drawn sketches by leveraging deep learning networks. The key research contributions of this paper include:

- We show how data visualization code generation from a hand-drawn sketch can be modeled as an image captioning problem, allowing the application of current state-of-the-art approaches in deep learning.
- The results demonstrate that Transformer models can achieve 95% structural accuracy in correct source code generation for hand-drawn sketches.
- The paper presents a novel approach for combining a Domain-Specific Language (DSL) and style transfer to generate training data automatically. The proposed solution overcomes the challenge of sourcing a data set of paired hand-drawn sketches with data visualization source code. Moreover, the results show that models trained on this synthetically generated data effectively generalize to hand-drawn sketches.
- The problem description enumerates the challenges of adopting existing evaluation metrics from related work and applying them directly to this Sketch2Vis problem. We propose new evaluation metrics to score generated visualizations that consider both structural and decorative features in the generated visualization.
- We compare and evaluate the performance of recurrent neural network (RNN)-based and Transformer-based networks on the Sketch2Vis problem. Our empirical results provide insights into what model architectures and domain-specific language (DSL) designs perform best on the Sketch2Vis problem.

The remainder of this paper is organized as follows: Section 2 discusses key challenges of building a model to translate hand-drawn sketches into multi-platform visualization code; Section 3 analyzes the feasibility of the Sketch2Vis problem in terms of dataset construction; Section 4 discusses different approaches for evaluating deep learning model performance on the Sketch2Vis problem and proposes new evaluation metrics; Section 5 analyzes empirical results from experiments we conducted on two key deep learning architectures: RNN and Transformer models; Section 6 compares related work with the techniques we explore in the paper; and Section 7 presents concluding remarks and lessons learned.

## 2 Research Challenges

Deep learning networks, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have shown great promise in understanding images and natural language. An interesting question is whether the performance they exhibit in other domains translates to the Sketch2Vis problem. This section discusses challenges specific to this problem that a deep learning model must address.

### 2.1 Challenges 1: Source Code is Platform-specific

A deep learning model should ideally be able to generate sketches using a variety of visualization libraries. However, source code for visualization libraries is typically platform-specific (e.g., Java vs. JavaScript, D3 vs. Tableau). To convert directly from sketch to data visualization, therefore, the machine learning (ML) model must be trained how to produce visualizations using the underlying visualization library to achieve the desired goal(s), i.e., the ML model must learn the syntax, grammar, and semantics of the underlying visualization tool. For example, if D3.js is used as the target visualization platform, the ML model needs to understand how to produce valid JavaScript code that will achieve a visualization corresponding to a sketch.

Although two data visualizations may appear similar to a human, if different programming languages or libraries are used to create them, the actual implementations can vary greatly, depending on the underlying platforms and programming languages. For example, the axis and plot type in a line graph are explicit to a human, regardless of whether or not a person has data visualization knowledge. However, different plotting libraries specify axes via different mechanisms, such as JavaScript arrays vs. Python lists. Users therefore need to produce separate implementations for the same visual representation each time a visualization is instantiated on a different platform and/or with a different programming language.

To train a deep learning model, a dataset is needed that associates (i.e., "pairs") images of sketches with the concrete source code to realize the appropriate visualization for the sketch. If the dataset is platform-specific (e.g., the source code targets a particular library), the model must be retrained for each individual target visualization platform. Moreover, variations in the target languages and frameworks may make this training process easier or harder.

### 2.2 Challenges 2: Sketches and Paired Source Code are Expensive to Obtain

Deep learning models require large volumes of data to increase their robustness and generalize effectively to as-yet unseen problems. Public datasets for most image processing problems typically contain over 100,000 training images and labels. For example, ImageNet [6] has 1,281,167 images with 21,841 labels and the Open Images Dataset [20] has 9,011,219 images with more than 5,000 labels.

To train a deep learning model to turn sketches into source code, a dataset that pairs hand-drawn sketches with the source code to instantiate the appropriate visualization is needed. There is no available dataset containing both hand-drawn visualizations and the corresponding source code representation. It is therefore hard to train and experiment with these models since producing a large dataset of paired sketches and source code requires working with data visualization experts, which is prohibitively expensive relative to other domains, such as labeling what is in an image (e.g., cats, dogs, flowers, etc.).

For example, 10,000 sketches and associated source code would be needed to train models on the low-end of data volume scale. Obtaining these types of datasets is not easily crowd-sourced. Moreover, even if the dataset *is* somehow crowd-sourced, ensuring that the source code labels are correct is a much harder problem than simply determining if a bicycle or street light is in a picture using the mechanisms (such as captchas) applied in related work.

### 2.3 Challenges 3: Correlations Among Human-drawn Visualizations

There is large variation in how a human may sketch a particular visualization. Each type of visualization must therefore be drawn numerous times in the training set for models to learn. For example, if we want a 20,000-image dataset supporting five visualization types, 4,000 images for each type of visualizations should be drawn repeatedly.

In contrast, human drawn sketches are affected by personal style, e.g., sketches drawn by same person are often similar, especially when the target visualizations are simple. In this case, even if the goals of data volume can be achieved, getting variation in drawing style still requires a very large number of human drawers. A quality dataset must, therefore, provide as much randomness as possible in the sample sketches.

### 2.4 Challenges 4: Evaluation of Generated Code

Deep learning models rely on automated scoring of their outputs to learn and improve throughout the training process. It is hard, however, to score data visualization quality automatically. The performance of generated natural language captions, which is the type of model we used to attack the problem, is often evaluated by n-gram matching metrics [1], such as Bleu [25] and METEOR [8]. Unfortunately, evaluating the predicted code snippets for a data visualization (which are the outputs of a deep learning model) focus on low-level details that may map poorly to visualization features.

In particular, the importance of different visualization features must be considered to enhance the usability of visualization implementations. For example, in a visualization code block, legend selections may contribute more than color selections to the validity of the visualization. Therefore, although many prior scoring mechanisms have been applied in prior natural language deep learning work, it is not clear if they are sufficient or effective on the Sketch2Vis problem.

## 3 Analyzing the Feasibility of Generating Data Visualization Source Code from Hand-drawn Sketches

Image captioning is the process of producing a textual representation of an image (e.g., a car parked in front of a copse of trees). In recent work, a number of problems have been phrased as image captioning, such as the generation of HTML for a sketch of a web page layout [3]. This section describes how we investigated the feasibility of phrasing the Sketch2Vis problem as an image captioning problem. It also explores a key challenge of obtaining data and explains how we overcame this challenge by developing a novel approach using a *domain-specific language* (DSL) and *style transfer*, which is a computational process that makes an image appear as if it was produced by a human artist rather than a computer.

### 3.1 Developing a Training Dataset

Preparing a large volume of hand-drawn images paired with source code to realize corresponding visualizations is hard, as described in Section 2.1 and Section 2.2. Style transfer has emerged as an active area of research. Significant progress has been made to advance style transfer capabilities, particularly in the domain of deep learning [22].

On the other hand, common off-the-shelf (COTS) visualization libraries (such as Matplotlib) have added simpler style transfer mechanisms to make charts look hand-drawn. Below we explore how we applied style transfer as a component of overcoming the challenge related to obtaining training data described in Section 2.2. In particular, we propose a solution that applies (1) a DSL (and its grammar) to generate source code for data visualizations randomly and (2) style transfer to convert these computer-rendered visualizations into images that appear hand-drawn.

Our dataset creation process inverts the normal dataset curation process of obtaining raw data and then labeling it with a human. Instead, the approach we employ operates as follows:

1. Generate the source code (label),
2. Execute the source code to render the visualization and export it as an image (semi-raw data), and
3. Apply style transfer and image augmentation techniques to produce human-like sketches of the visualization (raw data).

As discussed in Section 2.1, a key advantage of using a DSL is that it makes the training process independent of the target visualization library. Likewise, the language design can be tailored to facilitate faster learning. Compared with sketches hand-drawn by human artists, synthesized images that appear hand-drawn have several advantages, including:

1. **Scalable rendering of highly variable sketches.** A problem with a human-driven approach is that each human tends to produce sketches with a similar style. Using hand-made sketches to train not only requires a large number of hand-drawn images, but also a large number of people to ensure the dataset shows sufficient variation to generalize to the larger population. Our DSL-based approach enables a much wider variation in visualization features/types through randomization.
2. **Balanced training data distribution.** Since the dataset is created in a semi-random way, we can manipulate the distribution of the training data. Our approach avoids potential deep learning challenges caused by imbalanced datasets, which can yield models that generalize poorly.
3. **Dramatically lower cost.** A large volume of source code labels can be prepared through source code generation from the DSL without relying on time-consuming and costly manual creation of source code for each visualization. Hiring data visualization engineers to produce source code for sketches is prohibitively expensive and difficult to crowd-source.

## 3.2 From DSL to Visualization Code

As described in Section 2.1, a key problem in generating programming language code from sketches is that the models must be trained on each programming language that they target, which is suboptimal. To overcome this challenge—and to support realization of the sketch using multiple visualization platforms—we employ an intermediate DSL model that represents the abstract goals of the user with a simple syntax that can be learned readily by a deep learning model. The deep learning model produces code using our DSL and then code generators translate that DSL code into the implementation details of a specific visualization library.

Our DSL uses an XML-based syntax as shown in Figure 3. A token

```
<structure>
      <type> pie </type>
      <simplification> 0.2 </simplification>
      <ring> False </ring>
      <fillStyle> solid </fillStyle>
      <legend> True </legend>
      <legendPosition> left </legendPosition>
  </structure>
```

Fig. 3. An Example of Describing a Pie Chart Instance with the DSL Model.

dictionary (called a token pool) is built and updated for the DSL model, as shown in Figure 4. Visualization characteristics are described by tokens enclosed between starting and closing tags, such as ” $< type >$ ” and ” $< /type >$ ”. These tokens are categorical values tokens, such as the visualization type (e.g., scatter plot), indicating visualization feature candidates. A specification for a visualization is built from a sequence of tokens and the appropriate enclosing tags.

Images in a dataset are generated by different visualization tools. Therefore, the corresponding visualization features in our DSL may vary, i.e., features supported by some tool A may not be supported by another tool B. This syntax enables us to assign a visualization type with flexible features in our DSL (instead of a fixed length of features), such that each feature is independent. In this case, only supported features in the target visualization tool must be considered during dataset preparation.

As shown in Figure 4, after a DSL specification for a visualization is produced, we can apply a code generation adapter to produce the native visualization code needed to render the visualization on the target platform (e.g., matplot, D3, etc.). Adopting an intermediate DSL (rather than directly relying on native visualization code for the visualization specification) enables us to decouple the deep learning model from the generation of the source code.

Different code generators can be plugged into target different visualization platforms without retraining the underlying deep learning model. However, if a new platform has visualization capabilities that were not captured in the DSL language and trained tokens, additional training data is needed to support these new visualization features that were not trained on previously.

In summary, our DSL-based approach has the following key benefits:

- **Source code generators can produce visualizations for several platforms using a single model inference.** Since model outputs describe sketches independently of the concrete implementation, the deep learning model must only learn one syntax during the training process. A single inference from the model can be run through multiple code generators to target different platforms. The model and the source code generators can be shared and reused more easily.

- **The dataset can be updated and expanded easily.** When a training dataset must be enlarged to train more complex plots, new tags can be added to the token dictionary. After a model is trained that can interpret sketches and produce DSL instances, a code adaptor can be built for different target platforms without retraining the original models.

## 3.3 Dataset Construction and Expansion

To ensure high quality training images and reduce the possibility of overfitting, pairs of images and DSL instances in the dataset must be produced from multiple visualization tools. The goal is to ensure the rendered sketches of visualizations have different styles and wider variation that better match the variations in how humans sketch. Figure 4 shows these steps to initialize and extend the training dataset with new visualization tools.

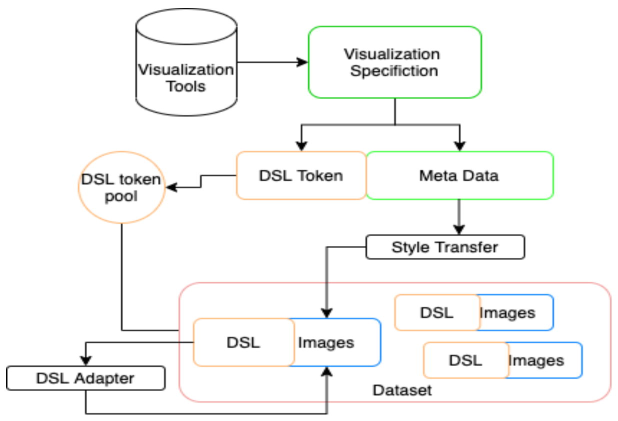


Fig. 4. Diagram of the Dataset Construction Mechanism.

## 3.4 Adding New Visualization Capabilities to the Training Dataset

Before a new visualization tool is used to generate images for the training dataset, the supported plotting arguments for the tool must be enumerated carefully. All data used to generate the training images must be stored for future use. This data is referred to as *metadata*, which represents the implementation of the native visualization tool and influences the design of an updated DSL by creating structural tokens and allowed values for the tokens, as discussed in Section 3.2.

Since the new tool's visualization *metadata* is likely different from the prior visualization tools used to create the training visualizations, the expected data formats and naming convention of visualization features may also be different. Figure 4 shows how a DSL token pool playing a role of word vocabulary is introduced to ensure a unified format for the DSL. Instead of generating new DSL tokens directly from the *metadata*, the new features must be compared with existing tokens in the DSL token pool and mapped to equivalent tokens wherever possible to eliminate semantic duplication across tokens.

By separating the *metadata* and DSL syntax, we can ensure data differentiation without introducing excess tokens and reduce the size of the word vocabulary, which is important for the deep learning models. Unused information in the *metadata*, such as the textual name for axes, can be saved for potential future use. For example, we can train the model to recognize and label text in sketches so that the model can communicate with data directly without requiring users to input this information manually.

## 3.5 Image Corpus Differentiation

To enhance the robustness of the synthesized hand-drawn image corpus—thereby overcoming the challenge we discussed in Section 2.4—we applied randomization to generate a widely varying corpus of images, including the following:

1. **Visualization type**. The types of visualizations (such as line plot, bar plot, and pie plot) are generated randomly.

2. **Specifications to the visualizations**. The parameters that are sent to the underlying visualization libraries for the visualization

type of visualization are randomized. For example, the parameters of a bar chart may include the height of the bars, color of the bar faces, color of bar edges, and legend position.

3. **Selections of the style transfer**. Style transfer approaches often have a wide variety of arguments that impact the rendering style. We currently achieve this capability by generating images from (1) multiple visualization tools, which support style transfer functions, and (2) Sketch style transfer deep learning models, which are trained based on real hand-drawn sketches of various instances.

4. **Text differentiation**. To ensure randomness in the textual elements in images, we randomize the labels applied to visualization elements. In the dataset that we experimented with, text in the images was randomly chosen from an English dictionary with different hand-drawn fonts and placed in varying positions on the chart that were within suitable bounds for the target text purpose (e.g., in a spot reasonable for an axis label to appear).

Using our randomization approach, a wide variety of sketches can be generated that appear hand-drawn and cover a large number of visualizations. Samples of our generated sketches are shown in Figure 5. Figure 5(a) depicts four visualization types generated via

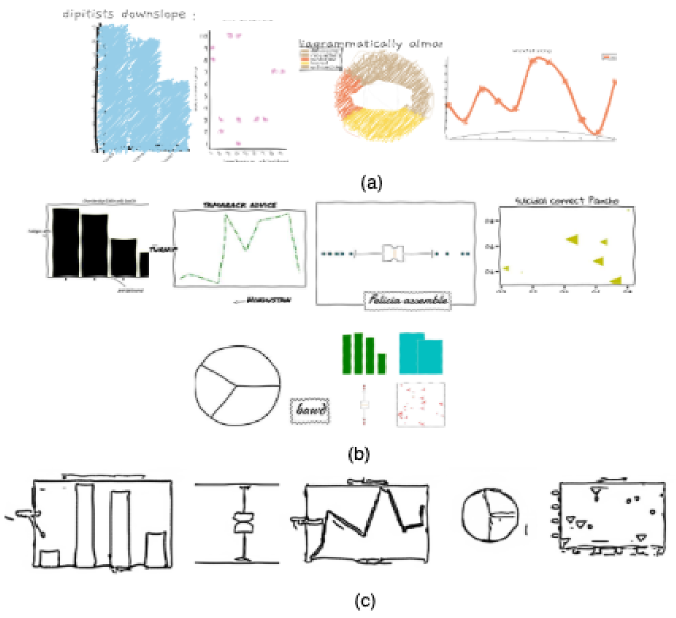


Fig. 5. Examples of Synthetically Generated Training Images Using Style Transfer

D3.js/RoughViz [39] on web browsers . Figure 5(b) shows five single visualization types and a visualization instance containing multiple visualization type generated via $XKCD()$ function supported by Matplotlib [16]. Figure 5(c) shows sketches transferred by *Photo-Sketching* [22] based on images generated by visualization tools.

## 4 Sketch2Vis Evaluation Metrics Exploration

A key question explored by our research was what deep learning architectures were most promising for the Sketch2Vis problem. RNN [2, 17, 37] and Transformer [44, 46] models have each shown excellent results on image captioning problems. We experimented with a number of RNN and Transformer architectures and present the best performing architectures that we found and their accompanying results on the Sketch2Vis problem.

One challenge of our architectural analysis was determining an appropriate scoring metric to use. We initially looked to establish metrics from the domain of machine translation since we were translating images into text. In translation problems, such as English to German translation, n-gram matching metrics [1] that split a string into n-length substrings have been widely used. For example, the Bleu score [25] is widely used in evaluation of translated sentences.

In contrast to natural language translations, n-gram matching is not well-suited to evaluate the generated DSL code since there are natural differences in contribution to code quality among different tokens. N-gram matching focuses on semantic similarities between the ground truth translation and the generated translation. In contrast, when evaluating the generated DSL code, execution must be considered since the DSL code must execute correctly *and* generate the desired visualization. For example, generated DSL code snippets that execute without run time errors should be considered more accurate than DSL code snippets with higher similarity scores that cannot be executed.

Previous work on natural language to code generation with deep learning considered both the output code blocks and execution output as metrics in the evaluation [45]. For example, prior work on GUI code generation [3] that used GUI screenshots as inputs evaluated models based on classification errors. That prior work evaluated a model's ability to correctly classify the GUI components in images.

The performance of natural language to SQL models has also been evaluated by the accuracy of the executed query results [45]. That approach reflected the model's ability to retrieve the correct data since even the generated SQL queries may be different from the ground truth.

Our approach presented in this paper evaluates models in the Sketch2Vis problem with one accuracy metric ($Acc_{cls}$) from prior research and two additional metrics ($Acc_{str}$ and $Acc_{dec}$) that we devised to overcome gaps observed when scoring generated visualizations. Each of these three metrics are described below:

1. **Classification accuracy.** $Acc_{cls}$ consider the results as a classification problem, calculating differences between the model output and ground truth DSL code. Outputs are penalized for producing tokens that differ from the ground truth and token sequences that differ in length from the ground truth. $Acc_{cls}$ can be calculated with Equation 1

$$Acc_{cls} = 1 - \frac{\sum(T_{false}) + \Delta(Len_{dsl})}{Len_{dsl}} \tag{1}$$

2. **Structural accuracy.** $Acc_{str}$ evaluates DSL mistakes that can result in structural errors, including syntax errors and incorrect visualization types. For example, a model that predicts the wrong visualization type or gives an invalid feature token to a certain visualization type will receive a low structural accuracy score, no matter how accurate the predicted DSL code is in other areas. $Acc_{str}$ can be calculated with Equation2;

$$Acc_{str} = \begin{cases} 0, & \text{if wrong semantic/plotting type} \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

3. **Decoration accuracy.** $Acc_{dec}$ scores the model's ability to understand local visualization features in the images, such as the desired color of lines or positions of legends. In other words, we want to evaluate if the model can choose the correct decoration tokens after generating the DSL structure. $Acc_{dec}$ can be calculated with Equation3.

$$Acc_{dec} = \frac{\sum \hat{T}_{dec}}{\sum T_{dec}} \tag{3}$$

## 5 Experimental Results

This section performs an initial exploration of which model architectures performed best on this problem formulation. Empirical results are presented from experiments we conducted on two key deep learning architectures: RNN and Transformer models.

| Visualization Tools | Plots | #Meta Parameters | #DSL Tokens | Other Augmentations |
|---|---|---|---|---|
| Matplotlib | Line | 4 | 51 | |
| | Bar | 7 | 28 | |
| | Box | 5 | 9 | layout |
| | Scatter | 3 | 46 | roughness |
| | Pie | 6 | 21 | text |
| RoughViz+React | Pie | 8 | 28 | text position |
| | Line | 9 | 20 | font |
| | Bar | 9 | 22 | |
| | Scatter | 8 | 15 | |
| Photo-Sketching | Line | \ | 10 | |
| | Bar | \ | 10 | |
| | Box | \ | 10 | |
| | Scatter | \ | 6 | |
| | Pie | \ | 10 | |

Table 1. Sketch2Vis Dataset Single Plot Parameter Description

### 5.1 Dataset Statistics

The dataset used in our experiments with RNN and Transformer architectures was constructed from two visualization tools described in Section 3.3. We validated these tools with a smaller number of hand-drawn sketches. Table 1 shows the visualization features and augmentation methods we used to generate the dataset.

We used style transfer mechanisms to create images that looked hand-drawn via (1) visualization tools (such as *Matplotlib* [16] and *RougViz.js* [39]) and (2) *Photo-Sketching* [22], which is a style transfer deep learning model. We used Matplotlib to generate Line, Bar, Box, Scatter and Pie plots and *RougViz.js* to generate Line, Bar and Pie plots, as shown in Table 1. Deep learning style transfer was performed with *Photo-Sketching* on sources images generated by Matplotlib.

Matplotlib is a popular Python 2D visualization library that generates quality visualizations [16]. The project is widely used and the source code repository on GitHub has 10,800 watchers and 55,800 questions on StackOverflow. Matplotlib includes a ”sketch-style” function, $xkcd()$, which can convert any Matplotlib graph into a hand-drawn format. We used the $xkcd()$ function to generate hand-drawn style plots randomly using Python. For browser-based visualizations, we used *RougViz.js* to generate hand-drawn style visualizations with *D3.js*.

Unlike style transfer functions provided by visualization tools, deep learning mechanisms use a dataset containing hand-drawn sketches [22, 29] to learn to transfer a scene into a sketch. These models capture complex variations in how humans draw and can reproduce images that mimic these variations. We adopted *Photo-Sketching* [22] in our dataset to generate simple monochromatic sketches, which is a different style compared to the visualizations generated with Matplotlib.

Both source visualization tools (Matplotlib and *RoughViz.js* in our experiments) share the same mechanism for producing the randomized *metadata*. The number of meta parameters and tokens is determined by the variability in the underlying visualization feature. For example, compared to color-related DSL tokens that can have a wide range of values, a DSL token indicating if a Bar chart is horizontal has only two possible values: ”True” and ”False.”

*Photo-Sketching* focuses on preserving the contours of images. Deep learning transfer may lose some visual information when applied to a visualization. For example, colors or data point styles in the original visualization will be omitted by the Photo-Sketching style transfer model and only the core features, such as the visualization type are retained. Therefore, the size of the DSL token pool will be reduced correspondingly compared to sketches generated by visualization tools. In particular, the information in *metadata* may not be rendered in generated sketches.

As a result, our current dataset contains both *images with individually colored data series* and rich visualization information and *scrawled black and white sketches* with basic visualization features. Moreover, to help avoid overfitting during training, we also include images containing multiple visualization instances, which helped our models generalize better.

Table 2 shows the number of tokens in the current DSL files and their distribution, which corresponds to the usage of different visualization features in the training set. As discussed in Section 4, we categorized

| #Token | #$T_{dec}$ | #Avg $T_{dec}$ | Avg DSL length |
|---|---|---|---|
| 1718138 | 438460 | 6 | 22 |

Table 2. Sketch2Vis Dataset Description

our DSL tokens into structural tokens($T_{str}$) and decoration tokens($T_{dec}$). Table 2 shows there are 438,460 $T_{dec}$ in the dataset, which comprises 25.5% of all tokens. On average, every image contains 6 $T_{dec}$ and 16 $T_{str}$, which indicates that $Acc_{cls}$ will more likely be affected by $T_{str}$.

Figure 6 shows the distribution of detailed DSL tokens based on token types. This dataset has balanced decoration tokens, stemming

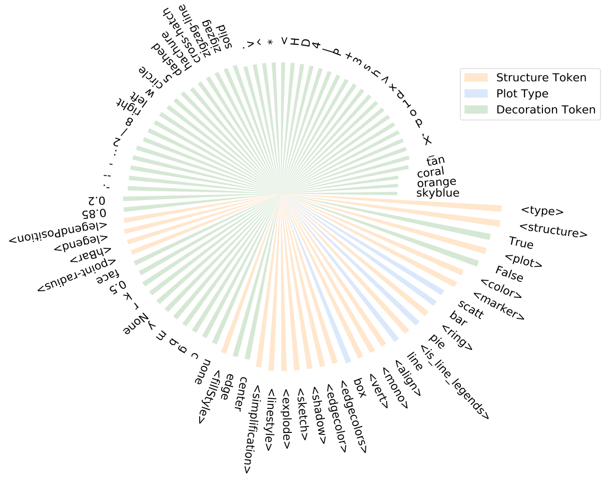


Fig. 6. Distributions of DSL Tokens in the Sketch2Vis Dataset

from the use of synthetic data generation, The dataset was split into training, validation, and test sets. The number of samples in each of these split datasets is shown in Table 3.

| #All | #Train | #Valid | #Test |
|---|---|---|---|
| 76942 | 63945 | 7105 | 5892 |

Table 3. Sketch2Vis Dataset Split Description

## 5.2 Deep Learning Architectures Explored

The Sketch2Vis problem requires an encoder-decoder architecture. In the encoder, image features are extracted from input sketches through a feature extractor, which is usually a convolutional neural network (CNN). Image features may be processed with a RNN for encoding in the latter part. The decoder portion of the model is then used to output sequences of DSL tokens.

Model construction in our experiments was based on different pairings of encoders and decoders. We constructed and compared our models based on two primary methods of processing sequenced data [7, 19, 24, 36], Recurrent Neural Networks (RNNs) and Transformers, as shown in Figure 7 and described below.

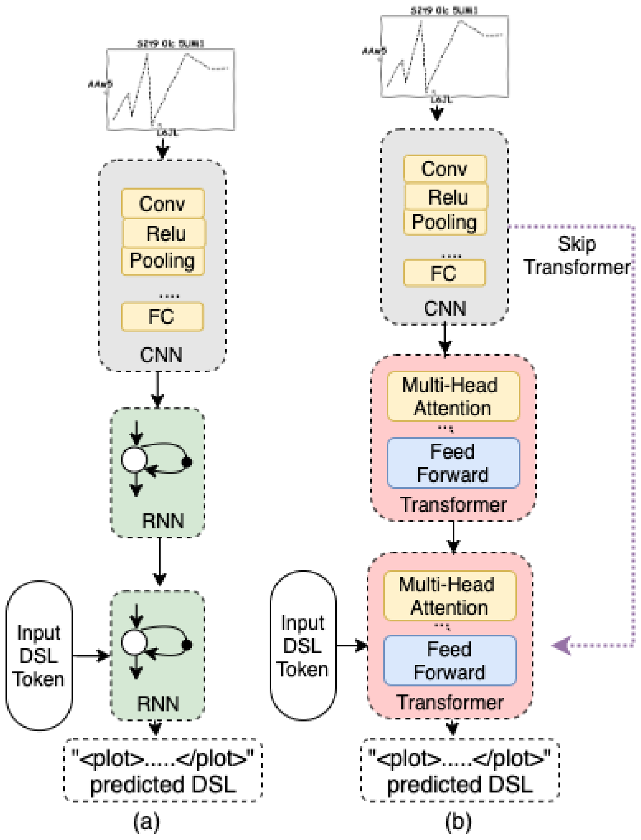


Fig. 7. Sketch2Vis Model Construction and Comparison

### 5.2.1 RNN Model Construction

Figure 7(a) shows the RNN model construction process.

At each timestamp in the encoder portion of an RNN the output from the image feature extractor is fed into recurrent units, which are cells used to process the token input at the $t_{th}$ step. At each step, $t_n$, the inputs consist of feature maps from the feature extractor, in addition to a sequence of the tokens produced in prior steps (e.g., $T_0 \ldots T_{n-1}$). A token at time t, $T_t$, is predicted from a list of previous tokens, $T_{t-i}, T_{t-i+1}, \ldots T_{t-1}$, where $i$ is the number of previous tokens in the DSL grammar. Each unit's inputs are therefore the aggregation of the image features and encoded tokens, as shown in the equation below:

$$T_t = Output(T_{t-i}, T_{t-i+1}, \ldots T_{t-1} | F_{image}) \tag{4}$$

The model keeps predicting the next token in the sequence until it reaches a terminating operator.

### 5.2.2 Transformer Model Construction

Figure 7(b) shows the Transformer model construction process. Encoded inputs are used to calculate a self-attention matrix to combine with attentions from output sequences. This approach allows models to generate outputs with context.

Transformer models have shown good performance on a number of natural language to natural language mapping problems, such as translation. Self-attention layers in the Transformer allow the model to get feature vectors of sequenced data without recurrent training steps, thereby enhancing training speed.

It is not clear, however, whether or not it is effective to use a Transformer to process the output of a CNN applied to a sketch image, which is not a sequenced input. In particular, prior research has not validated whether self-attention mechanism in Transformer encoder should be applied in features extracted by CNN. We next compare the performance between model architectures shown in Figure 7 to answer this question.

## 5.3 Model Architecture Comparison Results

**Model construction.** Table 4 shows empirical results from the different RNN and Transformer models that we built using different selections of encoders and decoders. In general, the CNN Encoder and Transformer Decoder models show promising results in generating visualization DSL code with the current dataset. Inaccurately predicted visualization instances were caused primarily by challenges in processing visualization features that make up a smaller amount of visual information in the sketches (*i.e.*, a smaller number of pixels), as well as visualization features with more options.

| Encoder | Decoder | $Acc_{cls}$ | $Acc_{str}$ | $Acc_{d}$ |
|---|---|---|---|---|
| CNN/RNN | RNN | 0.94 | 0.98 | 0.85 |
| CNN/Linear | Transformer | 0.95 | 0.97 | 0.87 |
| CNN/Transformer | Transformer | 0.77 | 0.74 | 0.60 |

Table 4. Experiment Evaluation Results of Encoder-Decoder Models

The Transformer architecture generally shows better performance than RNN architectures in the field of natural language processing [9, 36]. By comparing RNN and Transformer based models in our test set, however, we found that RNN-based models have similar performance on the Sketch2Vis problem. As a result, both models reached a score of 0.97 on structural accuracy, which means 97% of predictions are legal DSL expressions and the correct visualization type. **In other words, 97% of the predictions produced valid executable code that generated the correct visualization.** Decoration accuracy reaches 0.75, which means 75% of visualization decoration features are legal decorations for the visualization type and are predicted correctly.

Moreover, inspired by results in image captioning problems, we also adopted the Transformer as a part of an encoder to process output features of the CNN, as shown in Figure 7. However, our results show that image features encoded by a Transformer can hurt overall performance in the Sketch2Vis problem.

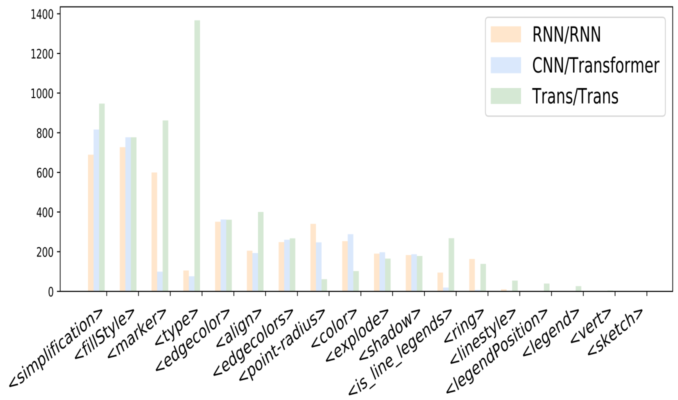


Fig. 8. Distribution of Mispredicted Decoration Tokens

**DSL Code Generation Performance.** Figure 8 shows the distribution of the incorrectly predicted visualization features in 3 models.

Errors in other visualization features are distributed more evenly and vary based on the size of the visualization feature's token vocabulary. The smaller a visualization feature's token vocabulary, the fewer errors the models make. For example, it is harder to predict the correct $< marker >$, which is the style of marker used for the data points in the visualization.

In contrast, $< is\_line\_legends >$ indicates if legends are visible in the visualization and performs much better since legends provide more information (e.g., cover a larger percentage of pixels) than the style of dots used for points. A trade-off exists between the model's usability (wider visualization capabilities in terms of the types of visualizations supported) and the model's accuracy. This trade-off shows the need to refine the DSL token vocabulary during dataset construction to only use tokens absolutely required for the needed visualizations.

The results in Figure 8 also indicate that we can combine the strengths of RNN and Transformer-based models into an ensemble to better support a wide range of visualization features and improve model performance. For example, a correcting mechanism can be appended to the model prediction stage, that weights both model's decisions.

Table 5 shows examples of the inputs and outputs in our experiments and the user-desired visualization. The "Training Sketch" shows an

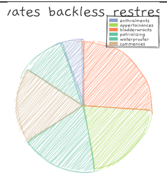
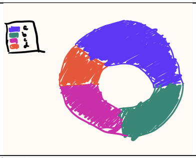
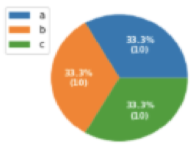
| | |
|---|---|
| Training Sketch |  |
| Validation Sketch |  |
| Generated DSL | <structure> <type> pie </type> <simplification> 0.2 </simplification> <ring> False </ring> <fillStyle> solid </fillStyle> <legend> True </legend> <legendPosition> left </legendPosition> </structure> |
| Generated Visualization |  |

Table 5. Examples of Model Input and Output

input created from *Rough.Viz* for encoder-decoder model training. The "Validation Sketch" shows a real hand-drawn sketch, where concrete text information is omitted. The "Generated DSL" is the predicted DSL code, which describes the visualization features, but is independent of the dataset specification (the data mapped to each axis, series, etc. can be plugged in separately). The "Generated Visualization" is a concrete visualization generated from our DSL using Matplotlib.

## 5.4 Results Discussion and Performance on Hand-drawn Sketches

In addition to evaluating on the generated test set using a combination of style transfer approaches in Section 5.3, as discussed in Section 5.1, we also prepared 100 hand-drawn visualizations drawn on mobile devices for validation purposes. Each visualization was manually labeled with its matching DSL code. Examples of hand-drawn sketches are shown in Figure 9, where half of the sketches are scrawled and the other half are colored.

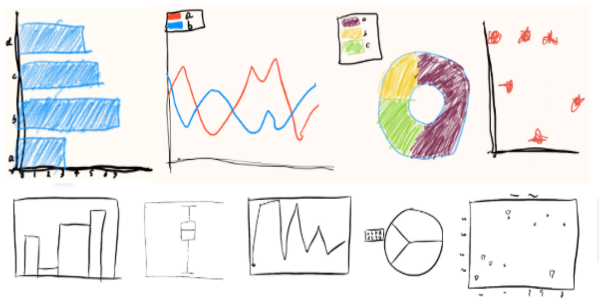


Fig. 9. Examples of Hand-drawn Sketches from Humans in the Validation Set

However, unlike images generated with style transfer mechanisms, where each sketch has an exact DSL description already determined, sketches drawn by humans are hard to evaluate with $Acc_{dec}$ since there may exist multiple valid DSL source code implementations. In this case, evaluation on $Acc_{dec}$ relies on user feedback. We currently only test $Acc_{str}$ on real hand-drawn sketch sets for validation purposes.

Since we include multi-visualization sketches in the training set, as discussed in Section 5.1, the model may predict DSL code containing multiple visualization instances, even if there is only one visualization instance in the image. If multiple visualizations are generated, users can select a visualization instance from the predicted options. For scoring purposes, when evaluating the hand-drawn sketches, any predictions containing the correct visualization DSL code are considered as correct in the calculation of $Acc_{str}$.

Table 6 shows the results of $Acc_{str}$ from the different models. The

| Encoder | Decoder | $Acc_{str}$ |
|---|---|---|
| CNN/RNN | RNN | 0.66 |
| CNN/Linear | Transformer | 0.95 |

Table 6. Evaluation Results on Hand-drawn Images

Transformer-based model shows the best performance in processing real hand-drawn sketches achieving 95% accuracy in identifying the visualization and correctly generating the correct code to implement the visualization. Although the RNN-based model performed well on the test set, it does not generalize well to hand-drawn sketches. This result indicates that the Transformer-based model is better at generalizing from style transfer images to real hand-drawn images.

**An important result from our analysis is that the $Acc_{str}$ on both the images generated using a style transfer approach and the hand-drawn images is close to the same (95% vs. 95%).** This resulting accuracy indicates that using style transfer to generate the training images is an effective approach for overcoming the challenge of obtaining human-labeled sketches. Since the Transformer models were able to generalize to hand-drawn data using synthetic style transfer data, future work can be significantly simplified by relying on a much larger corpus of synthetically generated data for training and smaller sets of human-labeled data for validation. Although we could not automatically evaluate the decoration accuracy (*e.g.*, colors, line style), $Acc_d$, our user feedback indicated that the decoration accuracy was similar in accuracy to the synthetic dataset (we are validating this result in future work with human studies).

Integrating more style transfer mechanisms into our current dataset is essential to enhance the model robustness. We also incur bias in our current evaluation on real hand-drawn sketches due to the limited number of drawers. To enhance the accuracy of our evaluations we are collecting sketches from more people, as discussed in Section 7.

## 6 Related Work

This section compares related work with the techniques explored in the paper.

**Data Visualization Efficiency**. As an emerging capability in data-driven applications, data visualization has drawn attention far beyond the visualization community, in nearly every domain, ranging from healthcare to smart cities. Due to the broad expansion of users, data visualization efficiency has become a critical requirement. Various applications have been designed to reduce the visualization learning curve and enhance usability for engineers who interact with visualization tools.

For example, FlowSense [43] allows users to construct dataflow diagrams from English sentences. Arjun [32] implemented a tablet-based data exploration tool with multimodal (i.e., pen, touch and voice) interactions. NL4DV [23] is a Python tool that helps developers create specifications by taking natural language(s) as input. Data2Vis [11] adopted Vega-Lite [30] as an interpreter to generate visualizations automatically from data specifications to visualization specifications. Qin [27] explored three paths to enhance efficiency in data visualization:

1. Improve the designation and input of visualization specifications,
2. Provide various approaches for data visualization, and
3. Automatically correct or refine users' generated visualization.

Our work builds upon achievements in path #1 and mainly focuses on path #2. We are also adding complementary mechanisms to our current model inspired by path #3. Our goal is to produce adequately detailed visualizations from hand-drawn sketches, thereby saving time producing the initial visualization implementation that can be later refined manually.

**Sequenced Output in Deep Learning.** Prior research on human-computer interactions via natural means (such as sketches) relied on enforcing strict constraints on users due to the challenge of processing informal human input by machines [18]. For example, earlier pen-based computer interactions required users to learn an unnatural way to draw sketch diagrams so computers can understand them [12, 21]. To overcome this challenge, shape recognizers were proposed to interpret pen strokes in human-drawn sketches and trained via a cluster-based approach that groups streams of pen strokes [18, 26]. Shape recognizers grant computers the ability to read users' drawing with more flexibility and can be extended by improving recognizers with additional types of supporting shapes.

Deep learning networks provide a solution for computers to extract features from natural human input, such as images, text, and voices [9, 14, 31, 34]. Beyond image classification, researchers attempt to use images to explain more complex human-comprehensible output, such as natural language expressions [15]. The encoder-decoder structure was adopted in the image captioning problem [37], thereby combining advantages of CNNs in processing images and RNNs in processing sequences of natural language expressions.

As an object detection mechanism, faster R-CNNs [28] were introduced for models to output captions in the context of classified objects. Transformers [36] further enhanced model performance and training speed by substituting RNNs as encoder and decoder. In contrast, the image-to-code problem shows significant compatibility with GUI components. For example, pix2code [3] was proposed to transfer UI screenshots of web applications into HTML components and was further extended to mobile app development [5].

## 7 Concluding Remarks

This paper analyzed the feasibility of generating data visualizations from hand-drawn sketches, a problem that we call Sketch2Vis. As shown in this paper, DSLs and style transfer approaches can be combined to help generate labeled sketches with associated source code. This combination helps to overcome the key challenge of scalably obtaining a dataset for training. Our experiments showed models capable of reaching 95% structural accuracy on hand-drawn sketches. In addition, the results show that synthetically generated images that use style transfer to make them look hand-drawn is an effective approach for generating a labeled training set.

The following is a summary of the lessons we learned from conducting the research presented in this paper:

- **Models can learn and generalize from synthetically generated sketches produced with style transfer approaches.** The models achieved an structural accuracy of 97% on this style transfer training set and 95% structural accuracy on the hand-drawn validation set – indicating that models can train on and generalize using synthetically produced sketches.

- **The CNN-Transformer structures performed best**. The results of on our experiments with RNN and Transformer models showed that a CNN-Transformer structure model had the best performance in processing input from multiple sources and generally demonstrated the feasibility of generating simple data visualizations from sketches. Even though more real hand-drawn sketches are needed for solid validation.

- **Labeling visualization instances with a DSL was a scalable way to address visualization automation problems with deep learning.** Dataset construction methods in our experiments showed how describing visualization images with DSL instances enabled deep learning models to extract features from single plot, and potentially allow users to create subplots.

Our future work is focusing on optimizing and extending our current dataset by integrating more visualization tools to enlarge sketch styles for the Sketch2Vis model. In our current work, we did not focus on processing textual information (such as variable names), and still use a manual mapping step to map these textual names to data series. We intend to work on automatic translation of axis and series labels into data queries in future work. We are also refining feature selections in our DSL model by investigating human demands and problems during the visualization process from online communities (such as StackOverflow) to enhance our model's usability by enlarging its supported features without greatly expanding the DSL vocabulary size. Finally, we are preparing more hand-drawn sketches for validation and designing a competition centered on our public data and evaluation metrics.

The code and data described in the paper are available in open source format.[1]

## References

[1] P. Anderson, B. Fernando, M. Johnson, and S. Gould. Spice: Semantic propositional image caption evaluation. In *European conference on computer vision*, pp. 382–398. Springer, 2016.

[2] P. Anderson, X. He, C. Buehler, D. Teney, M. Johnson, S. Gould, and L. Zhang. Bottom-up and top-down attention for image captioning and visual question answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6077–6086, 2018.

[3] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, p. 3. ACM, 2018.

[4] M. Bostock, V. Ogievetsky, and J. Heer. $D^3$ data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[5] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu. Automated cross-platform gui code generation for mobile apps. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*, pp. 13–16. IEEE, 2019.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

[7] Y. Deng, A. Kanervisto, and A. M. Rush. What you get is what you see: A visual markup decompiler. *arXiv preprint arXiv:1609.04938*, 10:32–37, 2016.

[8] M. Denkowski and A. Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pp. 376–380, 2014.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[1] https://blinded

[10] M. Diamond and A. Mattia. Data visualization: An exploratory study into the software tools used by businesses. *Journal of Instructional Pedagogies*, 18, 2017.

[11] V. Dibia and Ç. Demiralp. Data2vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks. *IEEE computer graphics and applications*, 39(5):33–46, 2019.

[12] M. J. Fonseca, C. Pimentel, and J. A. Jorge. Cali: An online scribble recognizer for calligraphic interfaces. In *AAAI spring symposium on sketch understanding*, pp. 51–58, 2002.

[13] M. Friendly. A brief history of data visualization. In *Handbook of data visualization*, pp. 15–56. Springer, 2008.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[15] M. Z. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CsUR)*, 51(6):1–36, 2019.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55

[17] X. Jia, E. Gavves, B. Fernando, and T. Tuytelaars. Guiding the long-short term memory model for image caption generation. In *Proceedings of the IEEE international conference on computer vision*, pp. 2407–2415, 2015.

[18] L. B. Kara and T. F. Stahovich. Hierarchical parsing and recognition of hand-sketched diagrams. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pp. 13–22, 2004.

[19] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pp. 67–72. Association for Computational Linguistics, Vancouver, Canada, July 2017.

[20] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, pp. 1–26, 2020.

[21] J. A. Landay and B. A. Myers. Sketching interfaces: Toward more human interface design. *Computer*, 34(3):56–64, 2001.

[22] M. Li, Z. Lin, R. Mˇ ech, , E. Yumer, and D. Ramanan. Photo-sketching: Inferring contour drawings from images. *WACV*, 2019.

[23] A. Narechania, A. Srinivasan, and J. Stasko. Nl4dv: A toolkit for generating analytic specifications for data visualization from natural language queries. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[24] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[26] B. Paulson and T. Hammond. Paleosketch: accurate primitive sketch recognition and beautification. In *Proceedings of the 13th international conference on Intelligent user interfaces*, pp. 1–10, 2008.

[27] X. Qin, Y. Luo, N. Tang, and G. Li. Making data visualization more efficient and effective: a survey. *The VLDB Journal*, 29(1):93–117, 2020.

[28] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pp. 91–99, 2015.

[29] P. Sangkloy, N. Burnell, C. Ham, and J. Hays. The sketchy database: learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics (TOG)*, 35(4):1–12, 2016.

[30] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[32] A. Srinivasan, B. Lee, N. Henry Riche, S. M. Drucker, and K. Hinckley. Inchorus: Designing consistent multimodal interactions for data visualization on tablet devices. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2020.

[33] tableau. Tableau. https://www.tableau.com/, 2019.

[34] S. Takaki and J. Yamagishi. A deep auto-encoder based low-dimensional feature extraction from fft spectral envelopes for statistical parametric speech synthesis. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5535–5539. IEEE, 2016.

[35] P. Utama, N. Weir, F. Basik, C. Binnig, U. Çetintemel, B. Hättasch, A. Ilkhechi, S. Ramaswamy, and A. Usta. An end-to-end neural natural language interface for databases. *arXiv preprint arXiv:1804.00401*, 2018.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[37] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE transactions on pattern analysis and machine intelligence*, 39(4):652–663, 2016.

[38] J. Walny, C. Frisson, M. West, D. Kosminsky, S. Knudsen, S. Carpendale, and W. Willett. Data changes everything: Challenges and opportunities in data visualization design handoff. *IEEE transactions on visualization and computer graphics*, 26(1):12–22, 2019.

[39] J. Wilber. roughviz.

[40] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*, 2017.

[41] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[42] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

[43] B. Yu and C. T. Silva. Flowsense: A natural language interface for visual data exploration within a dataflow system. *IEEE transactions on visualization and computer graphics*, 26(1):1–11, 2019.

[44] J. Yu, J. Li, Z. Yu, and Q. Huang. Multimodal transformer with multi-view visual representation for image captioning. *IEEE transactions on circuits and systems for video technology*, 30(12):4467–4480, 2019.

[45] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

[46] X. Zhu, L. Li, J. Liu, H. Peng, and X. Niu. Captioning transformer with stacked attention modules. *Applied Sciences*, 8(5):739, 2018.