Decompiling Data Visualizations from Images

Shreyas Kapur
shreyask@mit.edu

Moises Trejo
moisest@mit.edu

Josh Hilke
jrhilke@mit.edu

ABSTRACT
Creating static visualizations is a lossy process, a user's code and data is compiled in the form of a static image. These images are the default form of data visualizations, and are ubiquitous across both social media and academia. As visualization researchers, this lossy compilation into images makes it very challenging to experiment and tweak with design decisions of the original author. In this work, we present a novel deep-learning based algorithm that can *decompile* an image visualization, allowing researchers to easily edit and inspect its visual design. Our method and domain-specific language is more general than previous approaches and is extensible to more types of data visualizations without the need for expert-tuned heuristics.

Author Keywords
inverse graphics; chart recognition; optical character recognition, deep program synthesis

INTRODUCTION
Data visualizations are ubiquitous across both social media and academia. When making a visualization, designers make many decisions to such as picking colors, encodings, marks, axes, and labels.

Software packages like Tableau, ggplot [13] and matplotlib [5] allow for powerful creation of data visualizations, but their end product is in the form of an image.

We refer to this process of combining data and a design specification into a rendered image as *compilation*. This compilation is almost always lossy, the original code or the project file used to generate the images are rarely published alongside, despite the ever-growing ubiquity of using data visualizations on social media.

As data visualization researchers, if we want to then experiment and tweak with some of these design decisions, we typically have to try and reproduce the image from scratch before being able to manipulate it.

In this work, we present a novel deep-learning based approach to recover the design specification of a data visualization using only the raw pixels as input. Since the compilation procedure is lossy we do not attempt to recover the underlying data, and

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04. . . $15.00
DOI: https://doi.org/10.1145/3313831.XXXXXXX

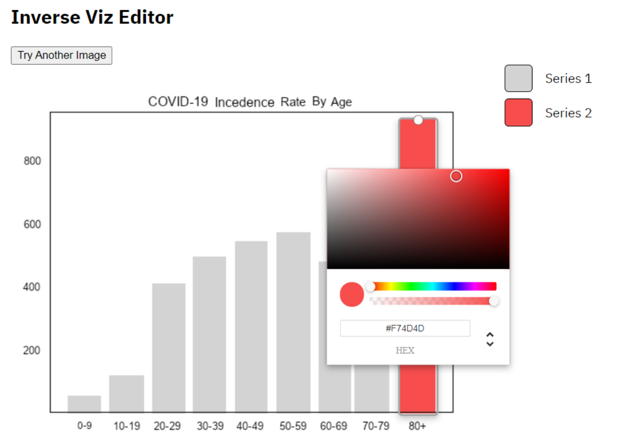

Figure 1. Screenshot of our editing program. The underlying data visualization was originally an image. Here we show the user editing the color of the last bar to draw attention to it.

focus strictly on the design specification. We also provide user-interface that can be used to manipulate the decompiled code in the form of a web extension.

As motivating examples, we show how our system can be used to make plots more accessible to color deficiencies, and be used to undo misleading design choices. Our prototype tool is available for use at http://inverse.firemeet.io/

RELATED WORK
The problem of derendering images into building blocks is known as inverse graphics. Kulkarni et. al. [8] used a convolutional variational auto-encoder based architecture to derender 2D graphics. Ellis et. al. [2] used a program synthesis approach to find programs for hand-drawings.

The application of inverse graphics specifically to data visualizations has also been studied before.

Kataria, Browuer, Mitra, and Giles [7] present an algorithm for extracting data points and text blocks from 2-D plots. Specifically, their tool extracts the X and Y axes, the values on the X and Y axes, labels, units, data points in the plot, and the legends.

Kanjanawattana, and Kimura [6] leveraged optical character recognition, natural language processing, and ontology to create a tool that extracts the X and Y axes information and chart data from line plots, bar graphs, and scatter plots. Data like the chart and axes titles and labels were parsed and entered into a data base which users could query to find visualizations related to topics of their interest.

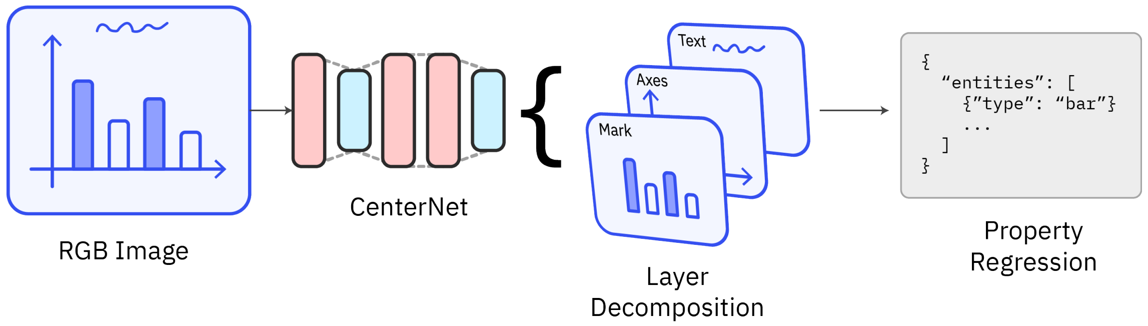

Figure 2. Overview of our decompilation engine. We modify the CenterNet architecture for object detection to simultaneously regress on bounding boxes and object properties. This results in a layer-wise decomposition into our DSL. The network is trained using a synthetic hand-designed prior to output a posterior distribution over promising programs.

Browuer et al. [1] address the challenge of extracting overlapping data points in 2-D plots, by applying simulated annealing, a stochastic method, to separate data points within a region of overlap.

Lu, Wang, Mitra, and Giles [11] implement a tool for extracting data points from a curved line on a 2-D plot. Data points for every curved line in the plot are extracted and stored in their own 2-D array. The goal of the tool is to provide a way to transform unstructured data like plots into structured data that can be stored in and queried from databases.

Zhou and Tan [15] offer a solution for detecting bar charts. Their system consists of preprocessing, detection and recognition stages, and uses a *Modified Probabilistic Hough Transform* algorithm to detect parallel line clusters which are used as a heuristic for identifying bars in a bar chart.

Previous work all commonly used hand-designed heuristics to extract meaningful information from images. Our work builds on the previous literature but also does not require designing careful expert heuristics for derendering, and is much easy to extend to new types of visualizations.

METHODS

The core of our system is the decompilation engine, which we will discuss first. Then we will discuss some of our design choices for the frontend UI to interact with our engine.

Domain Specific Language

Before decompilation, we need to first discuss our representation language for design specifications. We evaluated several existing schemes at different abstraction levels, like matplotlib [5] and Vega-Lite [12] as our main specification. Unfortunately these languages are not conducive to inverse rendering since they assume knowledge of the underlying data which is almost always impossible to recover. We want a design specifications that fully encourages a design vs data separation.

To that end, we design our domain-specific language (DSL) that is more akin to an underlying graphics program. Our DSL is embedded inside Python and factorizes a visualization into three layers: axes, encoding, and text that have cross integeration within the three. The DSL can be also be viewed as a JSON.

The text layer consists of free form text boxes and alongside text content. We did not implement font rendering and stuck to *Arial* for all our experiments. In the future, we would like to a generative model to embed glyphs in a manipulable latent space.

The axes layer specify the style of the axes lines, as well as the scale of the axes platform. The encoding layer specifies encoding entities such as lines, points, and bars, alongside their design properties. Placement of encoding entities is done the pixel space, and not in the data space.

We also allow the grouping of multiple encoding entities into different series such that the entities representing the same semantic data group can be manipulated simultaneously.

Neural Decompilation

Our neural architecture borrows from the CenterNet object detection network [14], since it allows us to predict both bounding boxes for all entities but also simultaneously regress continuous and discreet properties for each entity.

Taking inspiration from [3, 4, 9, 10], we treat our decompilation procedure as an inference problem, our neural network predicting a distribution over promising programs.

The network is then trained over a synthetic dataset generated by sampling different properties into rendered images, for which we have ground-truth labels for. Since we virtually have an infinite stream of data, it allows us to combat any overfitting issues in our pipeline. The network artifact is then used to decompile new images and the expectation of the mixture distribution is taken as the output.

The prior distribution data generation is rendered using matplotlib. We used 10^6 training images over a single epoch. The training was done using 4 Nvidia RTX 2080 cards over three hours. In practice we also generated distractors on the image sampled from the ImageNet dataset, including translation and flipping augmentations.

Optical Character Recognition

We treat the text layer as a separate layer from our decompilation. To identify text fields in an image, we use Google's Cloud Vision API which takes in images and returns pieces of text that it finds in the image along with a bounding box for that text. The bounding boxes identify where the text is on the image, and are used by the user interface to create editable text boxes.

User Interface
We implement our user interface as a web extension. Figure 1 shows our editor in action. After the extension is installed, the user right clicks on an image on a web page and then selects the extension. The image is sent to our OCR and machine learning engines running on a web server to identify fields that can be modified. These fields are sent back to the front-end/user interface which is a web page.

We pass data from an image that the user wishes to modify to our OCR and decompilation engines, which run on a web server. The user is then able to manipulate the decompiled representation, either in code, or using a visual editor. The editor can be used via drag and drop and sliders.

The server then ingests the modified code and recompiled the image back for display.

DISCUSSION
In this section we discuss applications of our tool and user feedback. Feedback primarily consisted of comments on existing features, potential new features, and ethical concerns. Potential new features are discussed in the Future Work section of this paper.

User Feedback
Our tool allows users to change the shape and height of bars in a bar chart, and to modify any text in the image. In general users enjoyed playing with these features, but felt that the functionality was a bit limited. Specifically, users wanted to use this tool on other types of graphs other than bar charts, which is the only type of visualization our tool could be used for in our MVP. For the final version, we modified this tool to work with scatter plots as well.

Applications
There are multiple use cases for this tool. Misspellings, data errors, or mis-labeled elements can now be corrected "on the fly" as opposed to the content creator having to produce and deploy a whole new version of the visualization. Additionally, adjustments to color can be made quickly and easily to account for accessibility issues like color-blindness.

Ethical Concerns
This can be used for bad so easily so we added a watermark The way we take apart visualization and give the user the ability to change these parts can be very useful to change and improve it for a specific group. Changing the way a visualization looks can include more people but it can also exclude people in the same way. We are mindful that if the general audience can edit a visualization then they have the freedom to exclude as many people as they want. Some examples of this are changing the title to be more confusing, changing colors to make it hard to discern the goal of the visualization, or changing the marks to lead to a different conclusion. Currently, anyone is able to change a visualization like this using editing software but we just made it so much easier to edit and share a visualization that was changed to be worse.

FUTURE WORK
In our user feedback, users request several additional features. While we weren't able to be address all of these requests, these are candidates for future work.

In the current implementation, components in the visualization like bars, data points, and text can be edited, but cannot be removed or added. In the future, we could implement functionality to further modify the visualization in this manner.

Our MVP focused on a small subset of visualizations. Specifically bar charts that were created using *Matplotlib*. For the final product we've expanded this subset to include scatter plots as well. Further work could be done to handle line charts, pie charts, and other types of visualizations. We believe this is the next step, and it is much easier to do given our new approach. All we have to do is define a generative model for the new types in Python.

In visualizations, text is sparsely distributed, making it challenging for traditional OCR systems to create precise bounding boxes that don't overlap with other elements. We had issues with OCR identifying and separating text on graphs – even while using Google's Cloud Vision API which is the state of the art. We believe there's room for improvement. In the future we could look for a different OCR system designed specifically for visualizations.

Lastly, users expressed concerns about the tool's potential applications in blackhat visualization generation. One way we could address this is to automatically add a watermark to any modified visualizations during the re-rendering stage of the image generation process.

REFERENCES

[1] William Browuer, Saurabh Kataria, Sujatha Das, Prasenjit Mitra, and C Lee Giles. 2008. Segregating and extracting overlapping data points in two-dimensional plots. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*. 276–279.

[2] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. 2017. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627* (2017).

[3] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2020. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381* (2020).

[4] Luke Hewitt, Tuan Anh Le, and Joshua Tenenbaum. 2020. Learning to learn generative programs with Memoised Wake-Sleep. In *Conference on Uncertainty in Artificial Intelligence*. PMLR, 1278–1287.

[5] John D Hunter. 2007. Matplotlib: A 2D graphics environment. *IEEE Annals of the History of Computing* 9, 03 (2007), 90–95.

[6] Sarunya Kanjanawattana and Masaomi Kimura. 2016. Extraction of Graph Information Based on Image

Contents and the Use of Ontology. *International Association for Development of the Information Society* (2016).

[7] Saurabh Kataria, William Browuer, Prasenjit Mitra, and C Lee Giles. 2008. Automatic Extraction of Data Points and Text Blocks from 2-Dimensional Plots in Digital Documents.. In *AAAI*, Vol. 8. 1169–1174.

[8] Tejas D Kulkarni, Will Whitney, Pushmeet Kohli, and Joshua B Tenenbaum. 2015. Deep convolutional inverse graphics network. *arXiv preprint arXiv:1503.03167* (2015).

[9] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. 2017a. Inference compilation and universal probabilistic programming. In *Artificial Intelligence and Statistics*. PMLR, 1338–1348.

[10] Tuan Anh Le, Atilim Giineş Baydin, Robert Zinkov, and Frank Wood. 2017b. Using synthetic data to train neural networks is model-based reasoning. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 3514–3521.

[11] Xiaonan Lu, J Wang, Prasenjit Mitra, and C Lee Giles. 2007. Automatic extraction of data from 2-d plots in documents. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 1. IEEE, 188–192.

[12] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350.

[13] Hadley Wickham and Maintainer Hadley Wickham. 2007. The ggplot package. (2007).

[14] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. 2019. Objects as points. *arXiv preprint arXiv:1904.07850* (2019).

[15] Yan Ping Zhou and Chew Lim Tan. 2000. Hough technique for bar charts detection and recognition in document images. In *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)*, Vol. 2. IEEE, 605–608.