Figure2Code: Enhancing Vision-Language Models with Synthetic Figure-Code Pairs for Improved Figure Understanding

Abhinav Lalwani
Stanford University
lalwani@stanford.edu

Johnny Chang
Stanford University
cjohnny@stanford.edu

Abstract

Advances in AI have improved how machines understand visual and textual content. However, understanding figures in documents remains challenging for vision-language models (VLMs). Figure understanding is crucial for models to learn from and interact with scientific, technical, and educational materials. Our project explores enhancing figure interpretation by fine-tuning VLMs with code-figure pairs. We introduce a method for generating a large synthetic dataset of figure-code pairs and define a comprehensive set of metrics for evaluating figure understanding, including MSE, CodeBLEU, Jaccard Similarity, and a new metric, HistDist. We benchmark our results using the LLaVA-7b and GIT models in zero-shot and fine-tuned settings. The LLaVA model displays promising results in generating accurate code syntax, achieving a CodeBLEU score of 0.7. However, the performance of the models in understanding the chart types and numbers in the figure is poor, indicating the need for further research on improving figure understanding in VLMs.

1. Introduction

Advances in artificial intelligence have steadily improved how machines understand the visual and textual content of documents. However, the rich data embedded in figures—graphs, charts, and diagrams—often eludes even the most advanced vision-language models (VLMs). While these models excel at parsing straightforward images or texts, they struggle to unpack the complex information presented in figures. This is demonstrated by the fact that the best performing VLM on the figure-question answering subsets of the latest vision understanding benchmark MathVista [13] is at 55%.

Figures are ubiquitous in scientific literature, technical documents, and educational materials, where they serve as crucial tools for delivering information in a more compact and visual approach. Unlocking this content could significantly enhance the way machines process and understand multifaceted documents, which would help with their ability to better assist in academic, scientific, and professional tasks.

Our project explores whether the interpretation of figures by VLMs can be improved by fine-tuning the models on datasets with code-figure pairs. Generating an accurate code requires the model to be able to understand all parts of the figure, including the format, values, categories, and background. Moreover, it would be possible to generate a large dataset for this task as given a code, we can obtain the corresponding figure by executing the code. Additionally, with a fine-tuned figure-code generation VLM as the feature extractor of the figure, we can pass the generated code as an intermediate representation of the figure to a more sophisticated language model for better reasoning skills. Our key contributions are two folds:

- We create the Figure2Code baseline and challenge datasets for benchmarking VLM's figure-to-code inference capabilities on three figure types.
- We fine-tune two VLMs on our datasets, which take in a figure as input and output predicted code for generating the figure.

Our datasets and code can be accessed on Hugging Face [1] [2]and Github [3].

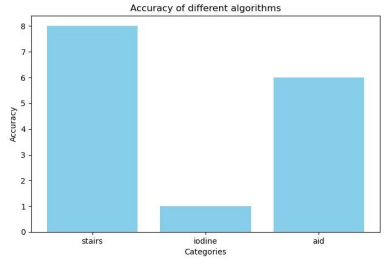
2. Related Work

Figure understanding is a critical skill for VLM's and has been extensively studied in the literature, formulated as a task in the form of figure question answering [11]. One of the most widely-used datasets is FigureQA [11], which has five figure types, such as line and pie charts, with

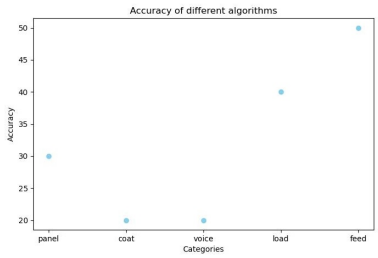
[1]Baseline Dataset: https://huggingface.co/datasets/abhinavl/figure2code_new_data_square

[2]Challenge Dataset: https://huggingface.co/datasets/abhinavl/figure2code_challenge_data_square

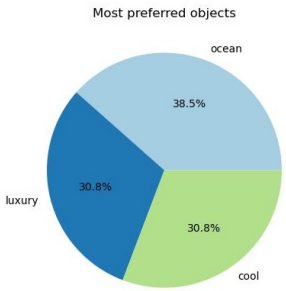
[3]Code: https://github.com/labhinav/Figure2Code



(a) Bar Chart

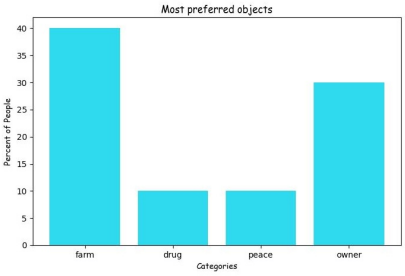


(b) Scatter Plot

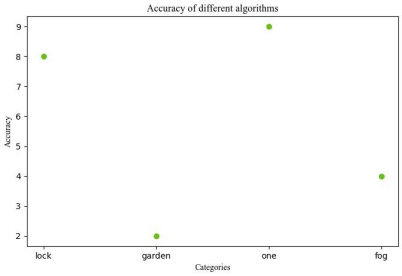


(c) Pie Chart

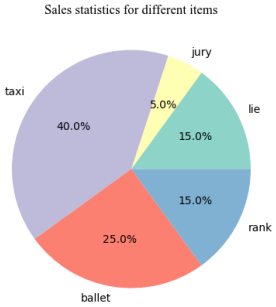
Figure2Code Baseline Dataset Figure Examples



(d) Bar Chart



(e) Scatter Plot



(f) Pie Chart

Figure2Code Challenge Dataset Figure Examples

Figure 1: Examples of Different Types of Figures in the Figure2Code Baseline and Challenge datasets

15 question templates that test the semantic understanding of relationships between the elements in the graphs. Other benchmarks on figure-question answering include ChartQA[14] with human written and generated question-chart pairs, LEAF-QA[4] with questions and charts created from real-world sources, and DVQA [10] that focuses on challenging bar charts.

The latest work from MathVista [13] combined all popular FigureQA datasets to create a unified benchmark, used by most state-of-the-art figure-question answering models. The best-performing figure-question answering models on the MathVista dataset include Qwen-VL-PLUS [2] and InternLM-XComposer2 [5]. Qwen-VL-PLUS uses the Qwen-7B large language model [1], the Vision Transformer (ViT) [6], and a position-aware Vision-Language adapter, trained on public image-text datasets. InternLM-XComposer2 uses a partial LoRA [9] approach where the LoRA parameters are only applied to the image tokens to maintain the language model's capability. However, the Qwen-VL-PLUS and InternLM-XComposer2 models achieve an accuracy of only 55.9% and 53.9% respectively on the figure-question answering problem subset, showing a big headroom for improvement in performance for the task.

As per our knowledge, no previous work has aimed to solve the task of figure-to-code translation. However, generating code from a graphical user interface has been studied extensively in several works. In pix2code [3], the authors use CNN and RNN to generate graphical user interface code from images, but the predicted outcome is a restricted one-hot encoding module for the UI's domain-specific language. In image2emmet [22], the authors use R-CNN and LSTM to convert UI images into HTML and CSS code. However, its empirical studies showed 60% precision in transforming UI components into code, showing opportunities to improve the model further. Soselia et al. [19] proposed a vision encoder and language decoder network to turn UI screenshots into HTML and CSS code, but its visual goal does not aim to extract numeric values and understanding from the input image. Hence, we believe that the task of translating figures to code would be achievable while valuable for the figure understanding task.

3. Methods

In our research, we address the problem of translating complex graphical figures into executable code, a task

we define as figure-to-code translation. The input to our method is an image of a figure, which can include a variety of graphical representations like bar charts, line graphs, scatter plots, or pie charts. The output is a piece of executable code that, when run, can regenerate the input figure.

In the following sections, we will describe our method for synthetically generating a large dataset of figure and code pairs to benchmark VLM for both fine-tuning and later evaluating model performance on figure-to-code capabilities. We will also describe the pretrained VLM models used for finetuning, including LLaVA-1.5 7B [12] and GIT 129M [21]. We chose these models for our task due to their widespread popularity and proven effectiveness in similar applications. The varying sizes of the models enable us to assess the impact that model size can have on our task.

3.1. Dataset Generation

We created two datasets, Figure2Code baseline dataset and Figure2Code challenge dataset, that consist of images of three figure types: bar charts, scatter plots, and pie charts. Each figure is paired with the code used to generate the images, and each type of figure has equal representation in the dataset distribution. To ground our generated figures with useful information for real use cases, we extract value, category, title, and value heading information from the DVQA [10] dataset. For each type of chart, we create a handwritten code template that uses the matplotlib library in Python. We fill in the extracted values from DVQA into the template to generate code samples in a randomized manner. For bar and scatter plots, we set the color of all images to 'skyblue' and pie charts with the 'plt.cm.Paired.colors' color scheme. For all the charts, we use the default matplotlib font, 'DejaVu Sans'. We then execute each of these code samples to obtain the corresponding images.

Through this approach, we generate the Figure2Code baseline dataset containing 10,000 unique code-figure pairs for each type of chart. To test the out-distribution performance, we create a Figure2Code challenge dataset which contains figures with unseen colors and fonts. We follow the same generation process as above to generate 1000 images per type of chart. However, for each image, the font is randomly chosen from 5 options. For bar and scatter plots, the color is selected by randomly generating a 6-digit hexadecimal string, which is the standard notation for color codes in HTML/CSS. For pie charts, we use a new color scheme, 'plt.cm.Set3.colors'.

Examples of images in the baseline and challenge datasets are shown in Figure 1.

3.2. Fine-tuning LLaVA

In our approach, we fine-tune the pre-trained VLM model LLaVA-1.5 7B, proposed by Liu et al. [12]. The architecture of LLaVA uses CLIP [17] as the vision encoder and LLaMA 7B [20] as the language model decoder. Concretely, the input images, 224 px by 224 px, are passed into CLIP to create image features, which are then projected to the word embedding space using a simple linear layer to produce the image embedding token. This image embedding token is then passed to the language model decoder alongside the input sentence to generate output text sequences.

The baseline LLaVA model has two stages of training, where the pretraining stage freezes the weights of both CLIP and LLaMA to train the linear projection layer between the image encoder output to LLaMA while the fine-tuning stage unfreezes all weights of the model to do instruction tuning.

For our approach, we are performing fine-tuning on LLaVA 7B using our figure-code pairs dataset. Each training pair uses a figure as the input and the code that generated the figure as the ground truth output. We build on the LLaVA-1.5 7B pre-trained baselline model from LLaVA's code base [8] by writing our own data pipeline, inference setup code, environment setup code (e.g. custom docker container).

3.3. Fine-tuning GIT

For our smaller model, we chose to fine-tune GIT 129M [21], a much smaller VLM than LLaVA. The GIT 129M architecture uses a CLIP image encoder and a single transformer-based text decoder to process both images and texts to output sequences of texts. Both models are jointly pre-trained on image captioning and fine-tuned on VQA2 [7]. Unlike more modern VLM architectures, GIT does not use a pre-trained decoder or perform instruction tuning. We utilized GIT's demo on huggingface [15] for model fine-tuning, adapting the preprocessing code and optimizing the hyperparameters.

4. Dataset Analysis & Features (0.5-1 pages)

For our experiments, we utilize the Figure2Code baseline and challenge datasets defined in Section 3.1. We randomly divide the Figure2Code baseline dataset into a train, validation and test split containing 24,300, 3000 and 2700 samples respectively. In the Figure2Code Challenge dataset, all 3000 samples are used solely for testing. Each image has a resolution of 100 dots per inch. As many CLIP encoders implicitly crop images to a square shape, we add whitespace to each generated image to make it square. This prevents valuable information in the x and y-axes from getting cropped off during training or inference.

The frequency of the number of labels per sample in the baseline dataset are displayed in 2. The vhallenge dataset also follows a similar distribution.

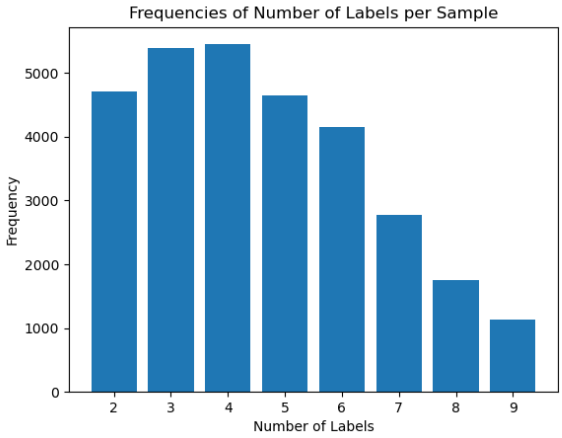


Figure 2: Label Frequencies on Figure2Code Baseline

5. Experiments

In this section, we explain our key evaluation metrics and the hyperparameters used for fine-tuning. We tested the baseline and fine-tuned models for both GIT and LLaVA on our Figure2Code baseline and challenge sets. To understand the effect of the size of training data and issues with overfitting, we additionally fine-tune the LLaVA model on a subset of the training data contain 1k samples.

5.1. Evaluation Metrics

5.1.1 MSE & CodeBLEU

For the overall results, we use two metrics: CodeBLEU [18] and mean-squared error (MSE). CodeBLEU evaluates the overlap between the generated code and the ground truth code by considering the surface match similar to the original BLEU [16] while also considering the grammatical correctness and the logic correctness of the code by leveraging the abstract syntax tree and the data-flow structure. We also report how much of the generated code has correct syntax and was able to execute successfully. The MSE score is calculated based on the pixel differences between the original images and the image generated by the generated code. The MSE metric would capture quantitative information as well as qualitative information about the figure such as background, style and fonts.

The MSE metric is sensitive to exact pixel alignment, which means minor variations in color, shading, or slight shifts in positioning that do not change the informational content can result in high MSE values, suggesting poor performance where the differences are not aligning with our goal of measuring figure understanding. Thus, we add additional metrics to capture whether the figure's metadata has been captured correctly. For measuring the accuracy of the values, we introduce a new metric called HistDist explained in Section 5.1.2

5.1.2 HistDist

Algorithm 1 Calculate HistDist

procedure HISTDIST($list1, list2$)
  $max\_val \leftarrow \max(list2, default = 0)$
  $m \leftarrow$ length of $list1$
  $n \leftarrow$ length of $list2$
  if $m > n$ **then**
    $list1, list2 \leftarrow list2, list1$
    $m, n \leftarrow n, m$
  end if
  $dp \leftarrow$ 2D list of size $(m + 1) \times (n + 1)$, initialized to ∞
  $dp[0][0] \leftarrow 0$
  for $i \leftarrow 1$ **to** $m + 1$ **do**
    $dp[i][0] \leftarrow dp[i-1][0] + |list1[i-1] - 0|$
  end for
  for $j \leftarrow 1$ **to** $n + 1$ **do**
    $dp[0][j] \leftarrow dp[0][j-1] + |0 - list2[j-1]|$
  end for
  for $i \leftarrow 1$ **to** $m + 1$ **do**
    for $j \leftarrow 1$ **to** $n + 1$ **do**
      $dp[i][j] \leftarrow \min(dp[i-1][j-1] + |list1[i-1] - list2[j-1]|,\ dp[i-1][j] + |list1[i-1] - 0|)$
    end for
  end for
  $ans \leftarrow dp[m][n]$
  if $max\_val = 0$ **then**
    $normalized\_ans \leftarrow ans$
  else
    $normalized\_ans \leftarrow \frac{ans}{max\_val}$
  end if
  return $\min(normalized\_ans, \max(m, n))$**end procedure**

We introduce a novel metric, HistDist, to measure the discrepancy between two histograms, as shown in Algorithm 1. Given that histograms such as bar charts may vary in the number of categories, traditional distance metrics such as L1 distance are not directly applicable. Additionally, set similarity metrics like Jaccard similarity are unsuitable due to their binary nature, where elements are either identical or entirely distinct while the similarity score is insensitive to the ordinal nature of histogram data. In histograms, proximity to the correct value often provides more meaningful insight than sheer equality or difference.

One potential approach is to utilize the L1 distance with zero padding to equalize histogram lengths. However, consider this example: if the target histogram is $[10, -100, 20]$ and the predicted histogram is $[10, 20]$, simple end-padding

would calculate the L1 distance as:

$$|10 - 10| + |-100 - 20| + |20 - 0| = 140.$$

A more accurate assessment would recognize the alignment of the first and last values and assign a distance of 100. This improved alignment can be achieved by strategically placing zero padding between the values in the predicted histogram instead of at the end.

To address such scenarios, we define HistDist as the minimum distance among all possible zero-padding configurations that preserve the sequence order of the histogram entries. This calculation can be efficiently performed using dynamic programming as shown in Algorithm 1.

Since the datasets may contain samples with widely varying magnitudes, we normalize each computed distance by dividing it by the maximum value in the target histogram to ensure comparability. Furthermore, to mitigate the influence of significantly inaccurate prediction outliers on the overall performance score, we clip the final score by the maximum length of either the predicted or target histograms.

Notably, HistDist is also applicable for comparing pie charts, as they are analogous to histograms where each segment denotes a percentage of the whole. This versatility makes HistDist a robust tool for diverse applications in data analysis. Intuitively, an average HistDist of N for a dataset can be interpreted as: the error in the prediction values is approximately equal to predicting N bars completely incorrectly for each sample.

5.1.3 Syntactically Correct Code

For all generated code files, we execute the files to test if the code executes without error. The syntactically correct code percentage is defined as the number of code executed without error divided by the total number of code generated.

5.1.4 Type Accuracy

To check if the model creates the correct type of figure such as bar graph, we extract the figure type keywords from both the generated code and the ground-truth code. The type accuracy percentage is defined as the number of correct type matches divided by the total number of code generated.

5.2. Hyperparameters and Fine-tuning Setup

For fine-tuning the LLaVA-1.5 7B model, we use a learning rate of 2e-4 and a cosine learning rate scheduler. We use the AdamW optimizer for the trainer. The model was fine-tuned on 1 epoch of the training data. To fit the model on a single A100 40G GPU, we use LoRA with a reduced batch size of 4.

The GIT model is fine-tuned on a single V100 GPU with the Adam optimizer. We use a learning rate of 5e-5 and a batch size of 8. The model is trained for 10 epochs and the checkpoint with the least validation loss is used for evaluation.

6. Results & Discussion

In this section, we present both qualitative and quantitative results along with error analysis of our baseline and fine-tuned models on both the Figure2Code baseline and challenge datasets. The final scores of all models are summarized in Table 1 for the Figure2Code baseline dataset and Table 2 for the Figure2Code challenge dataset.

6.1. Performance Analysis

From the results displayed in Table 1, we see that the LLaVA model fine-tuned on all 27K examples significantly outperforms the other models across many metrics. It achieved an MSE of 0.0163, a CodeBLEU score of 0.6809, a HistDist of 4.074, with 99.3% of the generated code being syntactically correct on the baseline dataset. Similarly for the challenge dataset, the fine-tuned LLaVA model outperformed all the other models on most metrics, achieving scores for CodeBLEU of 0.6247 and HistDist of 4.1673 while achieving 99.3% for syntactically correct code. Compared to the baseline dataset, the fine-tuned model performs slightly worse on the challenge dataset as it was out of distribution compared to the baseline it is trained on. Overall, this performance indicates that fine-tuning LLaVA on our dataset improved it's performance on figure-to-code translation.

The improved CodeBLEU and Syntactically Correct Code scores after fine-tuning indicate that the model gets better at predicting the correct code in terms of syntax, structure, and semantics. However, the reduced type accuracy indicates that the model may be forgetting the skill of predicting the correct type of chart. This may be occurring as the line of code specifying the type of chart is extremely small compared to the rest of the code, and hence the model may be optimizing for the syntax at the cost of predicting the correct type. However, the model improves the type accuracy compared to the LLaVA model trained on 1k samples, indicating that the model quickly forgets this skill but starts improving it again on further training. Even after fine-tuning, the HistDist score remains high, indicating Llava still has room for improvement in understanding the magnitudes of values in the figure. MSE scores do not correlate well with any of the other metrics, suggesting that MSE may not be a well-suited metric for this task.

Figure 3 shows the training loss for fine-tuning the LLaVA model on the 1k training set and full training set. The loss curves for both models show significant reductions in training loss observed within the first 10 to 30 training

	MSE	CodeBLEU	Syntactically Correct Code (%)	HistDist	Type Accuracy (%)
GIT	0.0256	0.422	**100**	4.645	33
GIT Fine-tuned	0.0256	0.422	**100**	4.645	33
LLaVA Baseline	0.0244	0.5344	54.4	5.1155	**63**
LLaVA Fine-tuned 1K	0.0169	0.539	61.6	4.76	39
LLaVA Fine-tuned All	**0.0163**	**0.6809**	99.3	**4.074**	46

Table 1: Final Performance Scores of All models on Figure2Code Baseline Dataset

	MSE	CodeBLEU	Syntactically Correct Code (%)	HistDist	Type Accuracy (%)
GIT	0.0323	0.3709	**100**	4.654	33
GIT Fine-tuned	0.0323	0.3709	**100**	4.654	33
LLaVA	0.0341	0.4531	78.7	6.4792	**57**
LLaVA Fine-tuned 1K	**0.0125**	0.448	54.9	4.689	46
LLaVA Fine-tuned All	0.0162	**0.6247**	99.3	**4.1673**	46

Table 2: Final Performance Scores of All models on Figure2Code Challenge Dataset

steps. This indicates that potential overfitting happens very quickly and the model quickly learns the syntax and structure of the code but fails to capture more nuanced information such as labels and the magnitude or proportion of the elements in the figure. The limited capability of the CLIP encoder likely contributes to this lack of understanding of text and element portion size, as it does not learn new features that were not present in its original pre-training data.

Comparing the results of LLaVA models trained on 1K samples and the entire dataset, we find that all metrics improve on fine-tuning with more data. Moreover, we find that the metric differences between the challenge dataset and the baseline are smaller with the model fine-tuned on more data, demonstrating that out-of-distribution robustness improves on fine-tuning with more data. This suggest that there may be further improvement by training for longer and on more data.

A surprising observation may be that the GIT baseline model and the GIT fine-tuned model both achieved perfect scores of 100% for syntactically correct code, as well as a lower HistDist score than the Llava Baseline. On manual inspection, we found that GIT blindly copies the example bar chart code given in the prompt for all samples. Fine-tuning does not cause this behavior to change but only leads to the model copying the prompt with higher confidence as shown in Fig. 3 where the loss decreases. Both GIT models achieved 33.333% accuracy in predicting the correct figure type, reflecting the one-third representation of bar charts in the dataset. The poor performance of GIT may be due to its smaller size or because the language decoder has no understanding of coding as it has only been trained in image captioning and VQA. Moreover, GIT may have lesser adaptability to new tasks due to the lack of instruction fine-tuning.

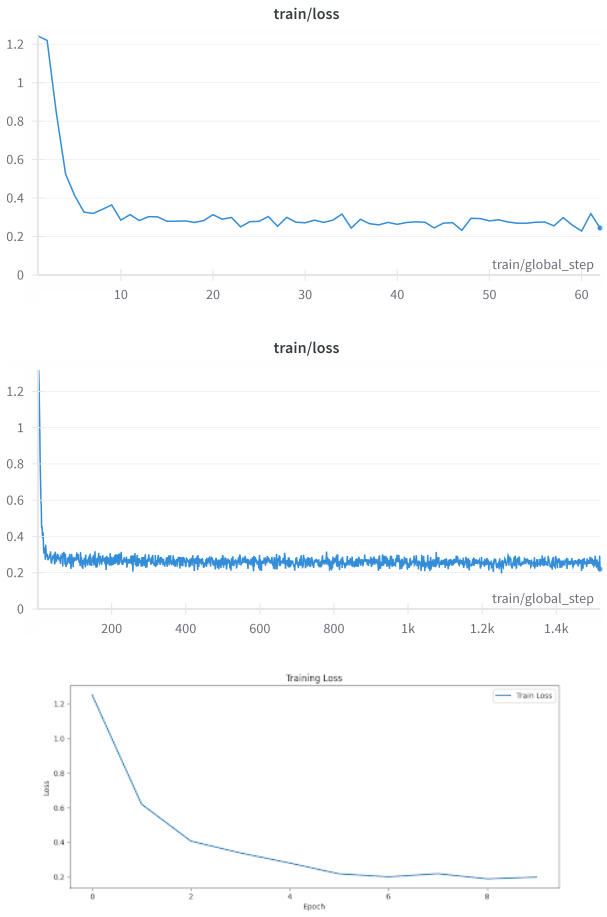


Figure 3: Training Losses for all models (Top: LLaVa-1K subset, Middle: LLaVa-all, Bottom: GIT)

The higher HistDist and lower Syntactically Correct Code scores of the LLaVA model compared to GIT may be explained by the LLaVA models' emergent capabilities. As described in a talk by Jason Wei from OpenAI [23], small models could perform better than medium-sized models on specific metrics even if the smaller model does not have more advanced skillsets. This is because smaller models might have simple skills that achieve well on a particular metric, in our case the GIT model copies and pastes the sample code, while medium-sized models are attempting to use an emergent complex skill that it is not proficient at, in our case, LLaVA tries to write syntactically correct code and understand the figures to generate the right code. One future exploration, as mentioned by Wei, would be to scale up the model to test if the emergent capability starts to improve.

6.2. Error Analysis

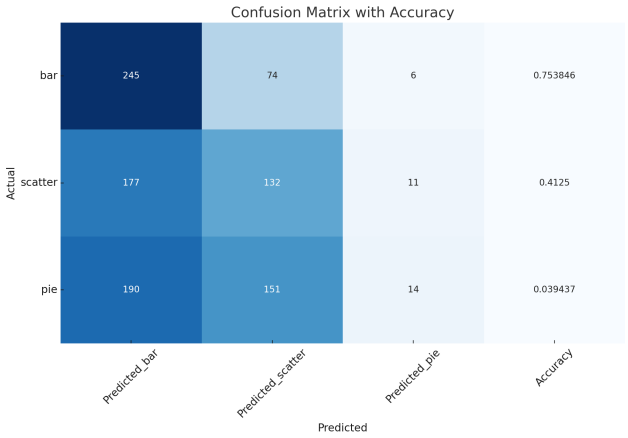


Figure 4: Confusion Matrix of the Actual and Predicted Figure Types with Generation Type Accuracies

Inspecting our best-performing fine-tuned LLaVA model's inference outputs, we noticed that the model's performance is heavily biased by the prompt, affecting its ability to correctly predict figure types. As shown in Figure 4, 61.2% of the generated graphs are bar charts, whereas only 3.7% of the generated graphs are pie charts. The higher probability of generating scatter plots than pie charts may be because the code to generate scatter plots is more similar to bar charts compared to the code for pie charts. This leads to the type prediction accuracy of bar charts at 75.38% being much higher than the other chart types.

An example of the output prediction from the fine-tuned LLaVA model on the challenge dataset is shown in Table 3. We can see that the structure of the code as well as the figure type are correct, but the values and the labels are incorrect. The ground truth code specifies the font and color of the bar, title, and axis labels, but this information was not captured in the predicted code. This suggests that the signals for font, text, numeric value, and colors are too weak, which means the fine-tuning has to either be done with a label-specific approach or with a much larger dataset and model.

Ground Truth Code

import matplotlib.pyplot as plt
# Categories and their corresponding values 	
categories = ['silver', 'hero', 'snow', 'pilot', 'show']
values = [60, 30, 50, 0, 80]
# Creating the bar chart 	
plt.figure(figsize=(8, 5)) Set the figure size (optional)
plt.bar(categories, values, color='99a0a8') # Plot the bars with a random color
# Adding title and labels 	
plt.title('Accuracy of different algorithms', fontname='Arial') # Add a title to the chart
plt.xlabel('Categories', fontname='Arial') # Label for the x-axis
plt.ylabel('Accuracy', fontname='Arial') # Label for the y-axis
# Display the chart 	
plt.show()

Predicted Code

import matplotlib.pyplot as plt
# Categories and their corresponding values 	
categories = ['faint', 'snow', 'pilot', 'cabin', 'sphere', 'basket']
values = [6, 8, 7, 9, 2, 7]
# Creating the bar chart 	
plt.figure(figsize=(8, 5)) # Set the figure size (optional)
plt.bar(categories, values, color='skyblue') # Plot the bars with skyblue color
# Adding title and labels 	
plt.title('Title') # Add a title to the chart
plt.xlabel('Categories') # Label for the x-axis
plt.ylabel('Values') # Label for the y-axis
# Display the chart 	
plt.show()

Table 3: A Sample Comparison of Ground Truth Code from the Figure2Code Challenge Dataset and Predicted Code of the Fine-Tuned LLaVA Model Trained on All Datasets

7. Conclusion & Future Work

In our study, we explored the task of figure-to-code translation by fine-tuning vision-language models (VLMs) using our synthetic dataset of figure-code pairs. We benchmarked the performance of two models, LLaVA-1.5 7B and GIT 129M, on both our baseline and challenge datasets. The LLaVA model, after fine-tuning, demonstrated superior per-

formance across most metrics, including CodeBLEU, and HistDist, while maintaining high syntactical correctness. This shows the effectiveness of large VLMs in understanding and generating code from complex figures while fine-tuning significantly improves performance on all metrics. However, it still struggles with understanding structural elements of the figure such as the type of chart and magnitude of values indicated by poor scores on type accuracy and HistDist. On the other hand, the GIT model struggled with the task, and was unable to do better than blindly copying the example code given in the prompt, even after fine-tuning.

Our research has a few future directions. One key direction is to expand our dataset to include a wider variety of figures and chart types to show the generalizability of our models. The expanded dataset would become a public benchmark dataset where other researchers can test their model's capability on figure-to-code translation.

To improve the model's performance on understanding figure components such as the type of chart and magnitude of values, we will include separate fine-tuning stages for these elements in future experiments. This will allow the models to learn these components without being distracted by having to learn the code structure at the same time. Additionally, we plan to experiment with larger VLMs on this task, hypothesizing that this could significantly improve the emergent capabilities. Finally, testing our model fine-tuned on the code-figure pairs with the figure-question answering tasks in the MathVista benchmark would validate our hypothesis that using code as an intermediate representation of charts could prove helpful for VLM reasoning skills.

8. Contributions & Acknowledgements

Johnny Chang worked on setting up cloud computing providers with multi-GPU training as well as the fine-tuning and inference of the LLaVA-1.5 model. Abhinav Lalwani worked on the dataset creation and metrics as well as fine-tuning and inference of the GIT model. Both members contributed equally to the literature review and writing of the paper.

Public repositories utilized:

- **LLaVA Repo**: `https://github.com/haotian-liu/LLaVA`
- **GIT Repo**: `https://github.com/NielsRogge/Transformers-Tutorials/tree/master/GIT`

References

[1] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[2] J. Bai, S. Bai, S. Yang, S. Wang, S. Tan, P. Wang, J. Lin, C. Zhou, and J. Zhou. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond, 2023.

[3] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot, 2017.

[4] R. Chaudhry, S. Shekhar, U. Gupta, P. Maneriker, P. Bansal, and A. Joshi. Leaf-qa: Locate, encode attend for figure question answering. In *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 3501–3510, 2020.

[5] X. Dong, P. Zhang, Y. Zang, Y. Cao, B. Wang, L. Ouyang, X. Wei, S. Zhang, H. Duan, M. Cao, W. Zhang, Y. Li, H. Yan, Y. Gao, X. Zhang, W. Li, J. Li, K. Chen, C. He, X. Zhang, Y. Qiao, D. Lin, and J. Wang. Internlm-xcomposer2: Mastering free-form text-image composition and comprehension in vision-language large model, 2024.

[6] 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.

[7] Y. Goyal, T. Khot, D. Summers-Stay, D. Batra, and D. Parikh. Making the v in vqa matter: Elevating the role of image understanding in visual question answering, 2017.

[8] Haotian-Liu. Haotian-liu/llava: [neurips'23 oral] visual instruction tuning (llava) built towards gpt-4v level capabilities and beyond.

[9] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models, 2021.

[10] K. Kafle, B. Price, S. Cohen, and C. Kanan. Dvqa: Understanding data visualizations via question answering, 2018.

[11] S. E. Kahou, V. Michalski, A. Atkinson, A. Kadar, A. Trischler, and Y. Bengio. Figureqa: An annotated figure dataset for visual reasoning, 2018.

[12] H. Liu, C. Li, Q. Wu, and Y. J. Lee. Visual instruction tuning, 2023.

[13] P. Lu, H. Bansal, T. Xia, J. Liu, C. Li, H. Hajishirzi, H. Cheng, K.-W. Chang, M. Galley, and J. Gao. Mathvista: Evaluating mathematical reasoning of foundation models in visual contexts, 2024.

[14] A. Masry, D. X. Long, J. Q. Tan, S. Joty, and E. Hoque. Chartqa: A benchmark for question answering about charts with visual and logical reasoning, 2022.

[15] NielsRogge. Transformers-tutorials/git at master · nielsrogge/transformers-tutorials.

[16] 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 on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[17] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision, 2021.

[18] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[19] D. Soselia, K. Saifullah, and T. Zhou. Learning ui-to-code reverse generator using visual critic without rendering, 2023.

[20] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023.

[21] J. Wang, Z. Yang, X. Hu, L. Li, K. Lin, Z. Gan, Z. Liu, C. Liu, and L. Wang. Git: A generative image-to-text transformer for vision and language, 2022.

[22] Y. Xu, L. Bo, X. Sun, B. Li, J. Jiang, and W. Zhou. image2emmet: Automatic code generation from web user interface image. *Journal of Software: Evolution and Process*, 33, 2021.

[23] YouTube. Stanford cs25: V4 i jason wei & hyung won chung of openai. https://www.youtube.com/watch?v=3gb-ZkVRemQ&list=PLoROMvodv4rNiJRchCzutFw5ItR_Z27CM&index=27, may 2024. Accessed: 2024-05-06.