D.I.A.G.R.A.M.: Development of Image Analysis for Graph Recognition And Modeling

Filippo Garagnani, Saverio Napolitano, Nicola Ricciardi
'Computer Vision and Cognitive System' course
Università di Modena e Reggio Emilia

***Abstract*—This report describes D.I.A.G.R.A.M., a system for diagram recognition and generation. We present its architecture, the logic behind deterministic algorithms and the training of its deep learning components. The project aims to transform visual diagram input into structured representations.**

I. Introduction

Diagrams are crucial in education, documentation, and many other fields. Developing a system that automatically understands and generates diagrams may reveal helpful for having high-quality, easily editable representations. This, however, poses challenges involving classification, shape detection, structure interpretation and symbolic representation. Our project proposes a modular architecture to tackle these tasks.

II. Related Works

The area of analyzing handwritten diagrams has been already studied in literature. Some key, modern, contributions are highlighted in this section.

Various works have already focused on the development, training and evaluation of specific deep-learning networks for extracting structures in arrow-oriented diagrams (like **Arrow R-CNN** [1]) or in vector diagrams (as in [2]). More traditional approaches tipically focus on specific diagram types and a restricted set of elements in them (e.g., **UML** [3]).

Some works in element recognition - like arrows - are particularly domain-specific (like medical imaging [4] [5]). Other studies focus on the so-called *online recognition* task, whose goal is to recognize diagrams as they are drawn (see [6], [7], [8]). The online recognition task quite often employs the use of relative strokes, and is mainly focused on flowchart diagrams.

III. System Architecture

Figure 1 illustrates the full pipeline, composed of four main modules: the Classifier, Extractor, Transducer, and Compiler. Each module is designed to process and transform the handwritten diagram image progressively toward a structured output.

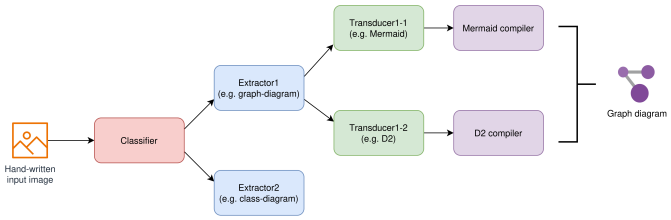


Fig. 1: Overview of the D.I.A.G.R.A.M. system architecture.

The Classifier is able to recognize the category of the hand-written diagram that the user submitted as input. This is necessary in order for the system to know to which Extractor pass the data. This module is able to apply object detection and semantic recognition to represent the diagram in a unified way. After that, the representation is sent to the Transducer, which converts the data in a Markup language of choice. The Markup language content is therefore sent to the Compiler, thus generating the high-quality and editable diagram in .png format.

A. Classifier Module

The Classifier is the first component of the pipeline. It is able to tell to which category a given image of an handwritten diagram belongs to. This is necessary in order to later know the Extractors that can process the input image.

B. Extractor Module

The Extractor is the key component of the D.I.A.G.R.A.M. system, able to transform the raw diagram image into a structured, category-agnostic representation. The Extractor recognizes the diagram's components, such as shapes, lines, and text, through an object detection network, and organizes them into a unified format. This representation serves as the foundation for the subsequent Transducer module, which converts the structured data into a domain-specific markup language.

C. Transducer Module

The Transducer is responsible for converting the unified, agnostic representation of a diagram into a domain-specific markup language. This transformation enables the subsequent Compiler module to generate high-quality visual outputs.

D. Compiler Module

The Compiler module is the final stage of the D.I.A.G.R.A.M. pipeline. Its primary role is to take the structured representation of the diagram, expressed in a markup language (e.g., *Mermaid.js*), and generate a high-quality diagram in a visual format such as PNG.

IV. Flowchart Diagram Recognition

A. Internal Representation

In order to pass data through the components of the pipeline in a unified way, it was necessary to decide upon an internal representation of the flowchart diagram. It consists of **elements** and **relations**. The latter are linked to at least one element. Every element has a *category* and possibly some *inner* and *outer text*. Every relation has a *category* too, either a *source* or *target element* or both, and possibly *text* related to it.

This internal representation is generated from the image provided by the user by the Extractor module, while it is parsed and converted to a markup language of choice by the Transducer.

B. Classifier Module

1) Preprocessing: Due to the fact that various kinds of diagrams' images could be submitted to the system, it was decided to streamline a unified preprocessing pipeline in order to reduce the differences that our classifier model had to handle. A few problems were analyzed and the final preprocessing pipeline for the classifier module is composed of the following operators, in order:

- **Gray Scaling**: In order to remove colors from images, which was deemed an useless information for classifying diagrams.
- **Otsu Thresholding**: To binarize the image and reducing the information to handle.
- **Median Filtering**: Some images in our dataset - and in the use-case - were taken on checkered notebooks. The application of a median filter, with kernel size equal to three, totally removed these.
- **Perspective Correction**: The diagrams' images, sometimes, didn't occupy the full photo frame. Finding four keypoints (if possible) and stretching the image helped to fully employ the frame.
- **Padding**: This was necessary in order to keep the same net for every image. Changes in resolution would have implied changes in the number of paramters of the MLP layers.

Figure 2 shows the step-by-step process of preparing an input image for the network.

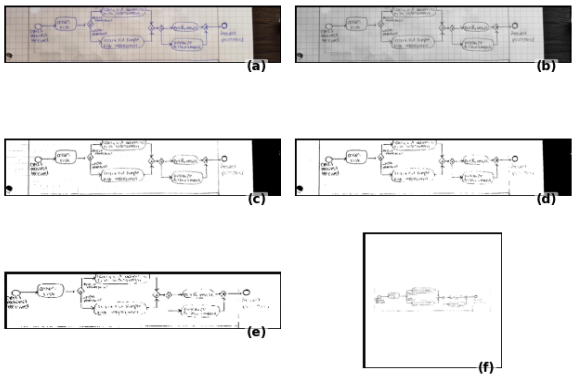


Fig. 2: The operations applied by the preprocessing module. **(a)**: The raw image, still colored. **(b)**: The image in gray scale. **(c)**: The binarized image, thanks to Otsu. **(d)**: The median filter result. Some previous thin lines are now no more. **(e)**: The image with a perspective correction. **(f)**: The image in a standard size, ready to enter the network.

2) Model: Since it was assumed that classifying two types of diagrams - with a third 'other' class - was a quite simple task, a small network was deemed enough. The model consists of three convolutional blocks - each containing a convolutional layer, followed by a batch normalization and a max-pooling layer - then followed by three linear layers (for a more detailed overview, see Appendix C). The output vector of the MLP contains three features - one for each class that this specific classifier needs to address. In order to determine the prediction, the softmax function is then applied. As later seen in section VI-A, a weighted Cross-Entropy loss is then used to train the model.

C. Extractor Module

The Extractor - despite its name - doesn't only need to *extract* objects from images, but also to find connections. Its pipeline can be seen at **Figure** 3.

1) Preprocessing: After some brief experiments, we found out that the object detection task through bounding boxes worked very well without any preprocessing technique. Moreover, the use of certain filters actually *downgraded* the performances - e.g. the use of aggressive median filters reduced the capacity of the OCR model for text extraction.

The only preprocessing applied is a gray-scale transformation. Other transformations are not deemed necessary - the model employed automatically resize the input image, so it's not needed to do it beforehand.

2) Bounding Box Detection: The model employed for object detection is a FasterRCNN, relying on a - as backbone - ResNet 50FPN [9]. The image can be passed rough, and the model automatically resize the image to its own resolution.
After having determined all the bounding boxes of the diagram image, in order to correctly create the diagram internal representation, there's the need to link together text,

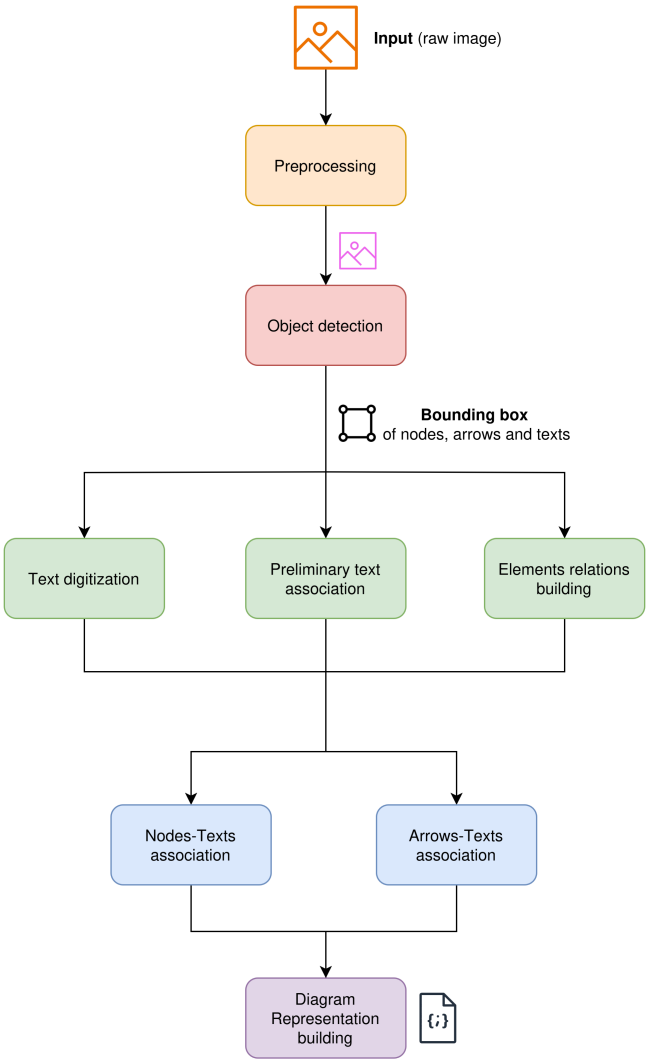


Fig. 3: The complete pipeline of the extractor module.

elements and relations. For more details about the model and its training, refer to **Section** VI-B.

3) Arrow Orientation: To later correctly assign arrows to their source and target elements, the need to determine where the tip of an arrow lies. This task involves a few problems, among which determining exactly the arrow shape and its end - because sometimes other pen signs may be found in the bounding box of the arrow itself. After a few trials (and errors; for further insight, refer to section VI-C), the approach yielding the best results was through the use of the object detection network (see section IV-C2) to recognize both arrow heads and tails.

4) Text Digitization: Since the task of converting an image of text into the relative string has already been solved in various ways, we settled upon the Microsoft *Trocr Handwritten* Transformer model; more precisely, the *base* version of it [10]. Every bounding box containing text that has been found is later passed to the model, and the resulting string generated by the model is linked to it.

5) Relationships Building: Once the text has been digitized, it's necessary to effectively compute which elements are connected by which arrows. This is done by finding, for each arrow, the elements nearer to the head (which we will assume to be the 'target' of the relation) and to the tail (the 'source' of the relationship exemplified by the arrow). These associations are nullified if the distance is above a certain adjustable threshold. This last step is done under the hypothesis that an arrow may come from nothing - or may point to nothing, having actually only either the source or the target.

6) Text Association: It can be assumed that every piece of text is linked to one and only one element or relation. So, every bounding box containing text is linked to the nearest element - it being either a relation or an element. The associations that require too long a distance are eliminated further in the algorithm, and the relevant text is discarded.

7) Element-Text Association: After that a text box has been assigned to an element, there's the need to understand if the text is either inside or outside the element and whether the text is actually referring to the given element or not. In order to do this, the overlap between the two bounding boxes is computed; if the overlap is over a certain threshold, heuristically decided, the association is kept. Otherwise, the distance between the two bounding boxes is computed - as the distance between their central point. If the distance is over a certain threshold, the association is kept; otherwise, it's discarded.

8) Relation-Text Association: After a text box is assigned to a relation, some computation is needed to fully understand how the text is associated to the relation itself. The relation's bounding box is then split into three different sections. The overlap between the text bounding box and each of the three sections boxes is computed - then, the section with the highest overlap value is then assigned to the text under consideration.

V. Dataset

Our approach leverages multiple handwritten diagram datasets to ensure robustness across different diagram types and drawing styles. We employed a strategic dataset partitioning scheme where different subsets serve specific model training purposes.

A. Dataset Sources

We collected data from seven diverse sources, encompassing various diagram types commonly found in technical documentation and educational materials:

1) **Finite Automata Dataset** [6]: Graph diagrams from the Czech Technical University collection, focusing exclusively on finite state machine representations.

2) **Flowchart Dataset (FCB)** [8]: Flowchart diagrams from the Czech Technical University repository.
3) **OHFCD_1 Dataset** [11]: Handwritten flowchart diagrams from the Computer Vision Center at UAB.
4) **hdBPMN Dataset** [12]: Business Process Model and Notation diagrams from the Data and Web Science Lab.
5) **Handwritten UML Class Diagrams** [13]: Class diagram collection from Kaggle.
6) **CircuitNet Dataset** [14]: Electronic circuit diagram repository.
7) **AI2D Dataset** [15]: Educational science diagrams from the Allen Institute for AI.

Many datasets don't include the task at hand - which is focused on recognizing flowchart and graph diagrams - but could be in future works. Moreover, many datasets not apt to our task were used to further enhance the capability of the classifier - letting it know if an uploaded image couldn't be further processed.

B. Dataset Usage Strategy

We implemented a targeted approach for dataset utilization based on the specific requirements of our architecture:

Classifier Training: To enhance model robustness and generalization capabilities, we employed class diagrams, circuit diagrams, and school diagrams from the UML, CircuitNet, and AI2D datasets respectively. This diverse selection ensures the classifier can effectively distinguish between different diagram types across various domains.

Extractor Training: For the specialized extraction tasks, we focused exclusively on the first three datasets: flowchart and graph diagrams from the FCB, OHFCD_1, and Finite Automata datasets. This targeted approach allows the extraction models to achieve high precision on the specific structural elements relevant to these diagram types, which were the core of our task.

VI. Experiments

A. Classifier

The first architecture explored for the Classifier module was a Convolutional Neural Network (CNN) following AlexNet's architecture, but smaller in size. It had two Convolutional layers, each followed by a batch normalization and then by a max pooling layers. The first layer had only 4 filters, while the second one had 8. The output of the second layer was then flattened and passed through a fully connected layer which downsized the features to 128, followed by a ReLU activation function. Other two fully connected layer brought the number of output features down to the number of classes. Then, a softmax activation function was applied to the output layer, in order to obtain the probabilities of each class. The model was trained on our own dataset, with only three possible output classes: 'flowchart', 'graph' and 'other'; it was trained for 10 epochs, with an AdamW optimizer (with a learning rate of 0.001) and a standard CrossEntropy Loss function.

We first tried such a small network because the features that had to be recognized were very simple, down to the basic shapes of the diagram elements. However, the results showed a non-decreasing value of the loss function of 0.9 - better than $-ln(3) \simeq 1.1$, which is random guessing, but still not satisfactory.

Judging this to be a problem in architecture size, we then tried adding another CNN layer with 16 filters; this one too followed by a batch normalization and a max pooling layer. The rest of the net was untouched; we also decreased the learning rate to 0.0001, in order to avoid overshooting the minimum of the loss function. The results, however, didn't show any kind of improvement, with the loss function still being around 0.9.

While analyzing the predictions, we came up with what was actually the problem: the dataset was heavily unbalanced, with the 'other' class being the most represented one, shadowing the other two classes. In order to solve this problem, we decided to use a weighted CrossEntropy Loss function, with the weights being inversely proportional to the number of samples in each class.

Algorithm 1 Cross-Entropy weight computation

Require: The classes' frequencies f_1, f_2, ..., f_c
$S \leftarrow \sum_{i=1}^{c} f_i$
$f_i \leftarrow (f_i/S)^{-1}$
$S \leftarrow \sum_{i=1}^{c} f_i$
$f_i \leftarrow f_i/S$
return f_1, f_2, ..., f_c

This final approach showed a significant improvement in the results, with the loss function decreasing to around 0.6 after 10 epochs of training, with a global accuracy score of 97% in the validation set.

B. Object Detection

For bounding box detection of diagram elements, we employed a Faster R-CNN architecture [9] with a ResNet-50 Feature Pyramid Network (FPN) backbone [16]. This two-stage object detection framework provides robust performance for multi-class detection tasks while maintaining computational efficiency suitable for diagram analysis applications. Its main components are:

Backbone Network: The ResNet-50 backbone serves as the feature extraction component, providing representations through its four residual blocks.

Feature Pyramid Network: The FPN enhancement augments the ResNet-50 backbone features by creating a top-down pathway with connections.

Region Proposal Network (RPN): The RPN generates object proposals by sliding a small network over the feature maps from the FPN. At each spatial location, the RPN predicts object presence probability and bounding box refinements for different scales and aspect ratios.

ROI Head: The second stage performs precise classification and bounding box regression on the proposals generated by the RPN. Feature vectors are extracted using ROI Align operations.

The original model was trained over the COCO dataset [17], which comprised 91 classes. It should be highlighted that the original training dataset was not in any way related to diagrams: with an efficient finetune, the network was able to swiftly change to our task. The finetuning is done with a learning rate of 0.005

The model employed was tested in three different ways: as zero-shot, after a 10-epoch training and after a 100-epoch training. The best-behaving one was, of course, the latter (to see the results of the different stages, see **Figures** 4 - 5 - 6 - 7). In Table I and Table II we show the Mean Average Precision and Recall of the model used.

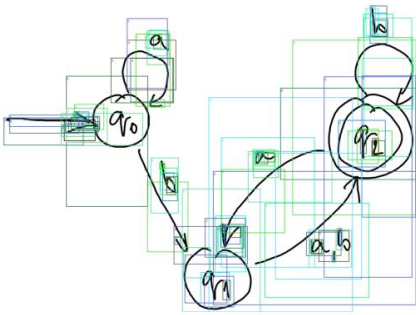


Fig. 4: The model employed in a zero-shot fashion.

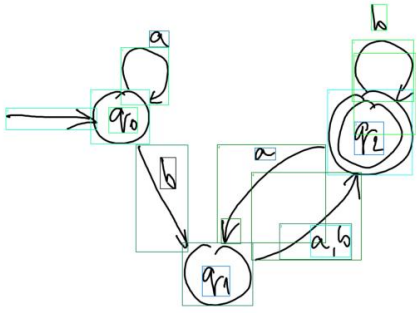


Fig. 5: The model employed after 10 epochs of training.

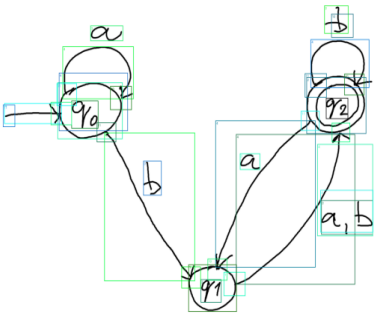


Fig. 6: The model employed after 100 epochs of training. More bounding boxes can be seen because we also added arrow's tails and heads.

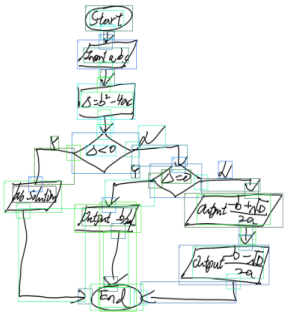


Fig. 7: The bounding boxes of a flowchart diagram, on the 100-epoch model.

TABLE I: Mean Average Precision (*mAP*) Metrics

Metric	Value	Description
mAP	0.8754	Overall mean Average Precision
mAP@0.5	0.9878	mAP at IoU threshold 0.5
mAP@0.75	0.9183	mAP at IoU threshold 0.75
mAP (Small)	0.6444	mAP for small objects
mAP (Medium)	0.8054	mAP for medium objects
mAP (Large)	0.8975	mAP for large objects

TABLE II: Mean Average Recall (*mAR*) Metrics

Metric	Value	Description
mAR@1	0.3436	mAR with max 1 detection per image
mAR@10	0.8877	mAR with max 10 detections per image
mAR@100	0.9128	mAR with max 100 detections per image
mAR (Small)	0.6631	mAR for small objects
mAR (Medium)	0.8670	mAR for medium objects
mAR (Large)	0.9141	mAR for large objects

Furthermore, we also think that the following metrics show a great precision:

- **Average X-axis error:** 6.84 pixels
- **Average Y-axis error:** 4.84 pixels

C. Arrow Orientation

As we've seen in the previous section, the need to find the exact position of an arrow's head arises (see **Appendix A and Appendix B** for a full overview of the first approaches).

The solution, as stated before, was to finetune the already working Faster R-CNN in order for it to also extract the bounding boxes of both tails and heads of the arrow.

D. Text Digitization

In order to choose an appropriate model, some tests were done over our specific dataset employing some different, premade models. The average results over the dataset are shown in Table III.

TABLE III: Text Distance Metrics

Model	Hamming	Cosine	Euclidean
microsoft-trocr-small-printed	1.392	0.150	7133
microsoft-trocr-small-handwritten	**1.346**	0.100	3947
microsoft-trocr-base-handwritten	1.549	**0.057**	**3003**

Some examples of the different models applied to our flowchart diagram dataset can be seen in Figure 8 and in Figure 9.

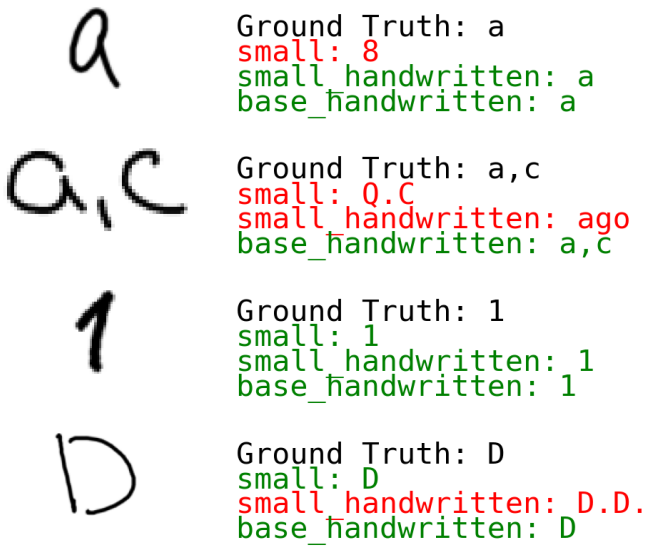


Fig. 8: Some random results of the text digitization over the proposed models.

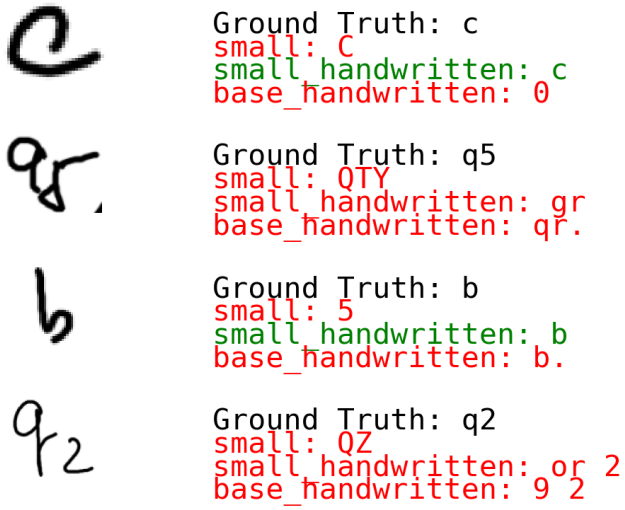


Fig. 9: Some results in which the *trocr-base-handwritten* fails.

VII. Discussion

The development of D.I.A.G.R.A.M. offers an insightful perspective on the integration of deterministic logic and deep learning for the interpretation of handwritten diagrams. One of the most valuable outcomes of this project is the successful design of a complete and functional pipeline (see Figure 10), which transforms raw visual input into a structured, editable format. This end-to-end workflow—from classification to rendering—highlights the feasibility of modular architectures for diagram understanding.

Among the modules, the Extractor proved particularly effective. Leveraging deep neural networks, it reliably identified diagram elements across diverse samples, validating the suitability of data-driven approaches for this subtask. Conversely, the Classifier, while operational, showed room for improvement, particularly in edge cases or less represented diagram categories. This suggests that the current training set may lack the diversity required for robust generalization, pointing to data augmentation and dataset expansion as immediate avenues for refinement.

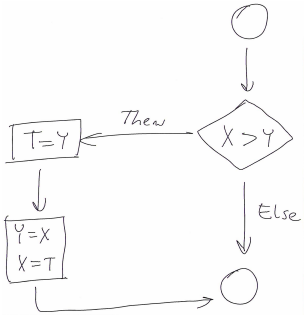
The system as a whole adheres to principles of scalability and generality. By decoupling the diagram type detection from the extraction and compilation logic, D.I.A.G.R.A.M. maintains flexibility in handling new diagram formats with minimal architectural changes. However, its performance remains sensitive to input quality and assumes a level of neatness in the handwritten diagrams that may not always be guaranteed in real-world scenarios.

In comparison to related work, D.I.A.G.R.A.M. distinguishes itself by targeting offline, image-based recognition (as opposed to stroke-based online methods) while retaining support for structural output generation. This positions it as a hybrid between traditional rule-based recognizers and modern neural approaches.

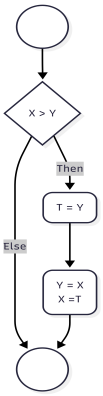
TODO: Nodo extra

VIII. Conclusion and Future Work

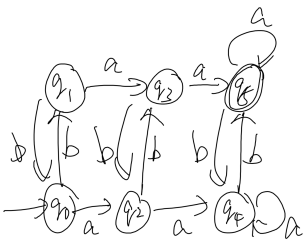
TODO



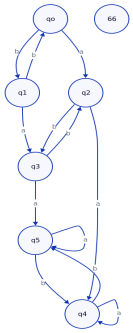
(a) An easy flowchart diagram.



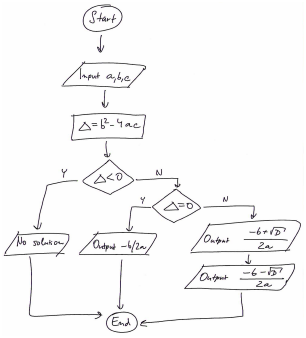
(b) The network's output.



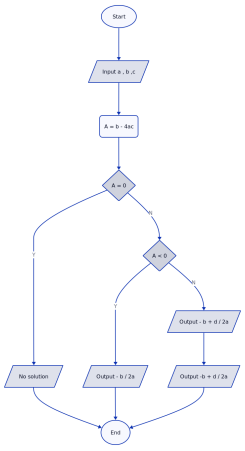
(c) A complex graph diagram.



(d) The network's output.



(e) A complex flowchart diagram.



(f) The network's output.

Fig. 10: A few results of our network.

REFERENCES

[1] B. Schäfer, M. Keuper, and H. Stuckenschmidt, "Arrow r-cnn for handwritten diagram recognition," *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 24, pp. 3–17, 2021. [Online]. Available: https://doi.org/10.1007/s10032-020-00361-1

[2] I. Saito, H. Yoshida, and K. Sakaguchi, "Sketch2diagram: Generating vector diagrams from hand-drawn sketches," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=KvaDHPhhir

[3] E. Lank, J. Thorley, and S. Chen, "An interactive system for recognizing hand drawn uml diagrams." 01 2000, p. 7. [Online]. Available: https://dl.acm.org/doi/10.5555/782034.782041

[4] K. Santosh, L. Wendling, S. Antani, and G. Thoma, "Overlaid arrow detection for labeling biomedical image regions," *Intelligent Systems, IEEE*, vol. 31, 11 2015. [Online]. Available: https://doi.org/10.1109/MIS.2016.24

[5] K. Santosh, N. Alam, P. Roy, L. Wendling, S. Antani, and G. Thoma, "A simple and efficient arrowhead detection technique in biomedical images," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 30, 04 2016. [Online]. Available: https://doi.org/10.1142/S0218001416570020

[6] M. Bresler, T. Phan, D. Průša, M. Nakagawa, and V. Hlavac, "Recognition system for on-line sketched diagrams," vol. 2014, 09 2014. [Online]. Available: https://doi.org/10.1109/ICFHR.2014.100

[7] M. Bresler, D. Průša, and V. Hlavac, "Detection of arrows in on-line sketched diagrams using relative stroke positioning," *Proceedings - 2015 IEEE Winter Conference on Applications of Computer Vision, WACV 2015*, pp. 610–617, 02 2015. [Online]. Available: https://doi.org/10.1109/WACV.2015.87

[8] ——, "Online recognition of sketched arrow-connected diagrams," *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 19, 09 2016. [Online]. Available: https://doi.org/10.1007/s10032-016-0269-z

[9] 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*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf

[10] M. Li, T. Lv, J. Chen, L. Cui, Y. Lu, D. Florencio, C. Zhang, Z. Li, and F. Wei, "Trocr: Transformer-based optical character recognition with pre-trained models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 11. AAAI Press, 2023, pp. 13 094–13 102. [Online]. Available: https://doi.org/10.1609/aaai.v37i11.26538

[11] B. Schäfer, M. Keuper, and H. Stuckenschmidt. (2021) Ohfcd - online handwritten flowchart dataset. https://tc11.cvc.uab.es/datasets/OHFCD_1. Accessed: 29/06/2025.

[12] B. Schäfer, H. van der Aa, H. Leopold, and H. Stuckenschmidt, "Sketch2bpmn: Automatic recognition of hand-drawn bpmn models," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science. Springer International Publishing, 06 2021, pp. 344–360.

[13] L. Piucco, E. L. Dos Santos, M. Johann, and L. M. Oliveira. (2021) Modelsketch: A tool for extracting uml class diagrams from hand-drawn sketches. https://github.com/leticiapiucco/ModelSketch. Accessed: 2025-06-29.

[14] A. Anthony, N. Lee, S. Sabri, and Z. Wu. (2021) Circuitnet: A dataset for hand-drawn circuit diagram recognition. https://github.com/aaanthonyyy/CircuitNet. Accessed: 2025-06-29.

[15] A. Kembhavi, M. Salvato, E. Kolve, M. Seo, H. Hajishirzi, and A. Farhadi, "A diagram is worth a dozen images," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 235–251.

[16] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie, " Feature Pyramid Networks for Object Detection ," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2017, pp. 936–944. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.106

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.

[18] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[19] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000. [Online]. Available: https://doi.org/10.1109/34.868688

[20] D. Deng, "Dbscan clustering algorithm based on density," in *2020 7th International Forum on Electrical Engineering and Automation (IFEEA)*, 2020, pp. 949–953. [Online]. Available: https://doi.org/10.1109/IFEEA51475.2020.00199

[21] N. Siddique, S. Paheding, C. P. Elkin, and V. Devabhaktuni, "U-net and its variants for medical image segmentation: A review of theory and applications," *IEEE Access*, vol. 9, pp. 82 031–82 057, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3086020

APPENDIX A
DOUBLE CLUSTERING

The first algorithmic approach proposed to solve this task - called *Double Clustering* - was as follows:

1) Apply a clustering algorithm to the bounding box.
2) Suppose that the biggest cluster belongs to the arrow; the other clusters are thought to contain useless elements for this task, and can be discarded.
3) Find the keypoints of the arrow using SIFT [18].
4) Find the two keypoints inside the arrow with maximum distance. Assume that these two points are closely related to the two ends of the arrow.
5) Run a template matching procedure inside the two clusters to find the head of the arrow.

For Step (1), we tried to employ both Spectral Clustering [19] (see Figure 11) and DBScan [20] (see Figure 12).

Fig. 11: The result of Spectral Clustering to one instance of an arrow bounding box in our dataset.

Fig. 12: The result of DBScan to one instance of an arrow bounding box in our dataset.

For Step (3), we used SIFT [18] in order to find keypoints and some descriptors for them - necessary for the later template matching. During Step (4), we were able to correctly deduce the overall direction of the arrow (see Figure 13).

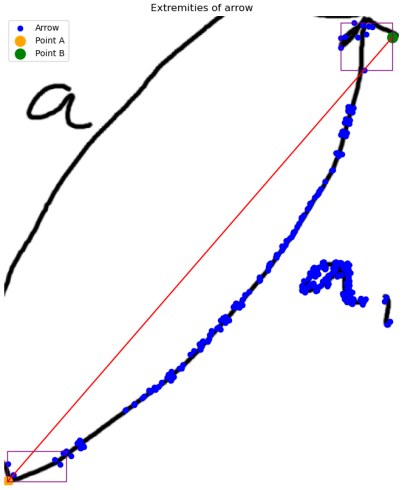


Fig. 13: The estimated direction of the arrow, finding the points that maximize the distance inside the cluster.

For Step (5), we had previously hand-picked some templates (see Figure 14).

Fig. 14: An example of hand-picked template with SIFT descriptors.

However, this approach didn't work as expected (see Figure 15). In particular, it never worked around self arrows: the maximized distance inside one of the two clusters is never between the tail and the head of the arrow.

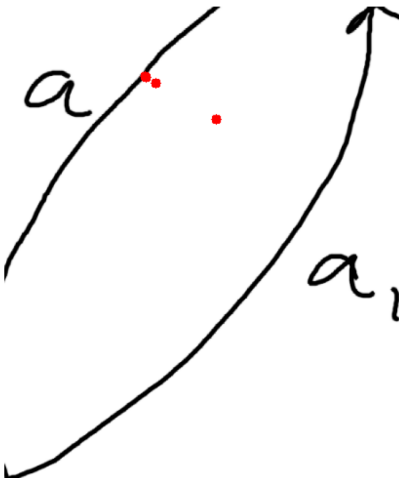


Fig. 15: The overall result of the algorithm.

APPENDIX B
ARROWNET

This second approach for determining an arrow's orientation was inspired by UNet [21]. The core idea was to segment an arrow's bounding box, and to use a specific CNN to try and determine the arrow's head point and the arrow's tail point. This downsampling-upsampling network is called **ArrowNet** and its specifics can be found in Table IV.

TABLE IV: ArrowNet Architecture Details

Stage	Layer	Out Ch	Operations
Encoder	Encoder1	32	Double Conv Block
	MaxPool	32	2×2 MaxPooling
	Encoder2	64	Double Conv Block
Bottleneck	MaxPool+Bottleneck	128	MaxPool+Double Conv
Decoder	Upsampling2	64	Transpose Conv Block
	Skip + Decoder2	64	Concat + Double Conv
	Upsampling1	32	Transpose Conv Block
	Skip + Decoder1	32	Concat + Double Conv
Output	Output Conv	2	1×1 Convolution

The entire architecture is composed of three main building blocks:

1) **Double Convolution Block:** Each double convolution block applies two consecutive 3×3 convolutions with padding 1, followed by batch normalization, and a ReLU activation after each convolution. This should preserve spatial dimensions while increasing feature depth.
2) **Upsampling Block:** The upsampling blocks use 2×2 transposed convolutions with stride 2, followed by batch normalization and ReLU activation.
3) **Skip Connections:** The architecture employs skip connections that concatenate feature maps from the encoder path with corresponding decoder features.

The two output float channels of ArrowNet should have been used to predict, in a given position: 1 if the head's point is in a certain position, 0 else; 1 if the tail's point is in a certain position, 0 else - for the first and second channel respectively.

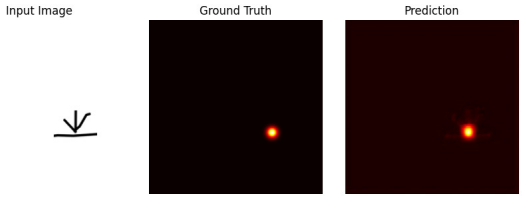


Fig. 16: Some results for arrow's head reconstruction.

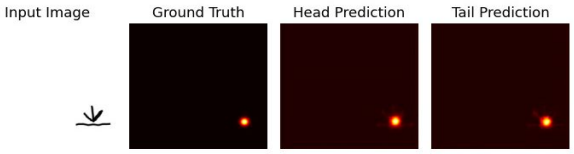


Fig. 17: The overall result of an ArrowNet input.

As seen in Figure 17, due to the low number of samples in the dataset, the network is unable to correctly distinguish between an arrow's head and tail.

APPENDIX C
THE CLASSIFICATION NETWORK

The **Classification Network** is a lightweight convolutional neural network, suited for processing grayscale images. The architecture is composed of convolutional feature extraction layers followed by fully connected classification layers. Some specifics about it can be found in Table V.

TABLE V: Classification Network Details

Layer Type	Out Channels	Kernel Size	Pool Kernel Size
Conv	8	2	3
Conv	16	2	3
Conv	24	3	2
Layer Type	**Out Features**	**Droput Rate**	**Activation Function**
Linear	2048	0.2	ReLU
Linear	128	0.2	ReLU
Linear	3	0.2	ReLU