

Article

Evaluating the Conformity to Types of Unified Modeling Language Diagrams with Feature-Based Neural Networks

Irina-Gabriela Nedelcu and Anca Daniela Ionita * 

Automation and Industrial Informatics Department, Faculty of Automatic Control and Computers,
National University of Science and Technology Politehnica Bucharest, 060042 Bucharest, Romania;
irina.nedelcu@stud.acs.upb.ro

* Correspondence: anca.ionita@upb.ro

Abstract: This article investigates the application of a deep learning model for evaluating the conformity of model images to types of UML diagrams to be used in self-training and educational settings. Our approach leans on a feature-based dataset that captures a broad range of modeling elements from class, state machine, and sequence diagrams, enhancing the ability to recognize a larger variety of categories selected for this research. The neural network trained with these features representing parts of the UML concrete syntax demonstrates 90% in classification accuracy on average, in respect to our previous research on UML diagrams classification without using a feature-based dataset. This study concludes that a feature-based approach, combined with advanced neural network architectures, can improve the classification of such images, especially in edge cases where diagrams contain similar graphical details but the whole does not represent a UML diagram. For the given research, we obtained a 0.87 F1 score.

Keywords: UML; deep learning; feature-based datasets; education and training



Citation: Nedelcu, I.-G.; Ionita, A.D. Evaluating the Conformity to Types of Unified Modeling Language Diagrams with Feature-Based Neural Networks. *Appl. Sci.* **2024**, *14*, 9470. <https://doi.org/10.3390/app14209470>

Academic Editors: Yu Liang,
Wenjun Wu and Ying Li

Received: 20 September 2024

Revised: 11 October 2024

Accepted: 11 October 2024

Published: 17 October 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Diagrams represented in Unified Modeling Language (UML) are mainly used in the design and documentation of software systems, serving as a fundamental step in the development process. However, with the growing complexity of software, the diversity of UML diagram types, and the potential for syntax errors, there is a need for automated systems that can effectively classify and verify these diagrams. The development of such systems not only enhances the efficiency of software engineering processes, but also supports educational initiatives by providing tools for learning and assessment. Recent studies have emphasized the importance of accurate UML diagram classification in improving software quality and supporting the educational framework of software engineering programs [1]. In the dynamic landscape of software development, an accurate classification and analysis of various UML diagrams holds importance given the current educational context [2]. It could also assist in reverse engineering by generating UML diagrams from legacy systems for better understanding. In Model-Driven Development (MDD), the model can verify that UML diagrams adhere to design patterns and system requirements. The model may also be useful in compliance verification by ensuring that designs meet regulatory standards, and in Agile development, where it can provide rapid feedback to validate changes in system designs during iterations.

The technique applied here is feature-based classification in machine learning [3], involving the utilization of a set of explanatory variables, features, to differentiate between various classes or outcomes. The efficacy of such models significantly relies on the quality and choice of features utilized. Concentrating on a specific subset of pertinent features can notably enhance the learning process for several reasons [4], such as decreasing the image size, speeding up the training process, or enabling focus on specific features that support distinguishing from one category to another.

Utilizing a subset of features diminishes the dimensionality of the problem, mitigating the curse of dimensionality and enhancing model performance. Models trained on a relevant subset of features are less prone to overfitting to noise in the training data, resulting in superior generalization to unseen data. Employing visually smaller-sized features can expedite training times and reduce computational expenses, thereby enhancing overall efficiency. A reduced number of features can enhance model interpretability [5], facilitating a clearer understanding of how input variables influence the output.

In essence, feature-based classification harnesses the predictive potential of individual features for data classification. Learning from a thoughtfully chosen subset of these features can refine the classification process by simplifying the model, mitigating overfitting, and enhancing efficiency and interpretability.

The ramifications of this advancement are profound. It enables a better examination of software design, facilitating improved predictions and insights into system functionalities. As we delve into the details of our enhanced model, we encourage readers to explore the nuances of this expanded approach and how we show edge case handling successfully against most performant tools available today.

Our prior work [6] established the groundwork by introducing a deep learning model that effectively classified class diagrams utilizing a feature-based dataset to evaluate existing implementations in the field [1]. This study introduces an extended approach that not only uses class diagrams but also introduces state machine and sequence diagrams. This enhancement goes beyond the addition of categories, and it signifies a comprehensive evolution of our deep learning framework. By incorporating state machine and sequence diagrams, we address a wider spectrum of UML diagram categories, capturing both the dynamic and static facets of system architectures. The augmented dataset now includes a diverse range of features extracted from these diagrams, empowering our model to learn and generalize with enhanced precision, similar to [7].

Recent advancements in neural network architectures have shown significant improvements in feature extraction and classification tasks through the use of attention mechanisms. For instance, ref. [8] demonstrated how attention-based models, originally designed for natural language processing, can be adapted to image recognition. By focusing on key segments of the input data, attention mechanisms improve model accuracy in handling complex patterns. This aligns with our approach of using feature extraction from UML diagrams to enhance classification accuracy, as attention mechanisms help the model focus on the most relevant parts of the input.

Similar work was partly performed in [9], which highlights the growing versatility of neural networks in complex prediction tasks. The focus of this research is on CO₂ emissions and it demonstrates how multi-head attention mechanisms can enhance performance in multi-objective scenarios.

The use of attention mechanisms, as explored in the above study, and other modern techniques such as transformer-based networks, have demonstrated significant improvements in image and feature classification tasks. This paper's use of a multi-head attention mechanism serves as a valuable reference for how feature extraction and focus on specific data segments can enhance model performance, aligning with our approach of extracting features from UML diagrams to improve classification accuracy.

In our study, the research focus is extended to classifying multiple UML diagram types, on feature extraction for improved classification. As the methodology, we focused on building neural networks following the multi-layer perceptron approach for our feature-based training data.

The current research investigates how a classification task performs on a feature-based dataset. This article explains how to build a UML diagram dataset and how to process the files, the images themselves, and the labels, including validation to confirm that a significant information has not been lost. Then, we propose the use of a neural network that works on a feature-based dataset to classify the entire image. All of these steps and the training process itself are detailed in Section 2. Section 3 describes the results, the metrics monitored

during training, and standard measures applied in these models, such as precision, recall, and F1 score. The results are compared with similar research achievements when aiming to classify UML diagrams and these are described in Section 4. This section also covers how the model works on edge cases that raise difficulties due to many similarities to other diagrams. Section 5 lists the conclusion of this research.

2. Method

The study considered multiple criteria (topic, methodology, dataset types, diagram types, performance metrics, key features), which are presented in Table 1 in comparison with several related works. This research gathered required input data from official documentation or other research papers, as listed in Section 2.1. Section 2.2 describes how we built a feature dataset. Section 2.3 explains how files were processed to all be consistent and to match certain patterns, to simplify their use in the training phase. Section 2.4 presents the data augmentation techniques that were applicable to this project and the procedure used to apply these techniques. Section 2.5 lists the steps we followed to process the images in the format required before training and how we performed data validation to ensure no significant loss was present after conversion. Similar to the images, the labels for each category also had to be processed, and that is described in Section 2.6. The last part, Section 2.7, explains how we define the model's architecture, training the model as well as evaluating performance.

Table 1. Research comparisons between similar papers and personal investigation.

Research	Gosala et al. [1]	Tavares et al. [2]	Rigou et al. [3]	Hwang et al. [4]	Current Article
Research topic	Automatic classification of UML class diagrams	Supporting software engineering education through UML diagram classification	Discovery of UML class diagrams from textual specifications	Semantic feature adjustment for image segmentation	Classification of UML diagrams for educational and self-training purposes
Methodology	Convolutional Neural Network (CNN)	Various CNN architectures for classification	Deep learning approach focused on feature extraction	SFA-Net: Advanced neural network architecture	Feature-based neural networks
Dataset Type	Full image diagrams	Full image diagrams	Text-to-diagram processing	Image segmentation in remote sensing	Feature-extracted UML images
Diagram Types	Class diagrams	Class, sequence diagrams	Class diagrams	Not focused on UML	Class, sequence, state machine diagrams
Performance Metrics	Accuracy, precision	Accuracy	Recognition accuracy	Segmentation accuracy, feature representation	Accuracy, F1 Score, robustness through cross-validation
Key feature	Introduction of CNN for classifying UML diagrams	Emphasis on educational context for UML classification	Feature-based extraction from textual data	Enhanced feature adjustment techniques	Extended classification to multiple UML diagram types, focus on feature extraction for improved classification

2.1. Build the Initial Diagrams Dataset

As part of this project, the initial step was defining the classes to be identified. In our case, we focused on the following types of diagrams:

- Class diagrams;
- Sequence diagrams;

- State machine diagrams.

Given that UML diagrams are extended to a wider variety of types, we created a separate category for other UML diagrams and one category that contains images unrelated to UML diagrams. For the non-UML type, we mainly focused on other images containing diagrams, especially diagrams that look similar to UML.

To conduct training on a feature-based dataset, we had to with images containing the UML diagrams as actual sources specified in Table 2. From the whole diagram, we explain how we altered the initial dataset to obtain a feature dataset.

Table 2. UML diagrams sources.

Image Diagram Sources
Unified Modeling Language Specification Version 2.5.1 [10]
UML Diagrams Examples [11]
LibreSoftTeam Github database [12,13]
Google-Images-Search python script 1.4.7 [14]

- Unified Modeling Language Specification Version 2.5.1 [10]

This is the official documentation from The Object Management Group Standards Development Organization (OMG SDO) and it defines the specification for a graphical framework for visualizing, defining, building, and documenting the elements of distributed object systems.

- UML Diagrams Examples [11]

Utilizing UML for over a decade to model, analyze, and document software projects, the author recognizes its increasing complexity. To simplify understanding, one developed a website as a personal reference. While some found it helpful, the author acknowledges UML's ongoing complexity and occasional inconsistencies, noting the challenges in reporting issues to the standards body. Despite these challenges, they recognize the value of UML for experienced users, suggesting supplementary learning materials for newcomers.

- LibreSoftTeam/2016-uml-miner [12,13]

The 2016-uml-miner is a study conducted by the LibreSoftTeam to analyze the usage of UML (Unified Modeling Language) in Free and open-source software (FOSS) projects on GitHub. The project includes a series of scripts designed to mine GitHub repositories for UML files and to process and analyze these files to understand their usage in the development lifecycle of these projects. The project involves several key scripts:

- github-api.py—This script interacts with the GitHub API to retrieve a list of files from repositories and stores the tree structures in JSON format.
- umlfiles2table.py—This script processes a CSV file containing verified UML files and creates an SQL table with the UML file data.
- langs2sql.py—This script converts language data from JSON files into SQL commands to populate a MySQL database.

These scripts help automate the process of identifying, extracting, and analyzing UML files from a large number of GitHub repositories, providing insights into how UML is used in open-source projects.

The 2016-uml-miner project on GitHub is described in the conference article [12] presented at the 2017 IEEE 14th International Conference on Mining Software Repositories (MSR). This article details the creation and analysis of a comprehensive dataset of UML models found in various GitHub repositories. The study provides insights into how UML is used in open-source projects and the types of UML diagrams commonly found. The dataset and analysis tools, such as the scripts included in the 2016-uml-miner repository, were pivotal in facilitating this research.

- Google-Images-Search 1.4.7 [14]

In order to utilize this library, it was necessary to enable the Google Custom Search API, generate API key credentials, and setup a project:

- i. Create a new project in Google Cloud.
- ii. Enable the “Custom Search API” for the project.
- iii. Generate API key credentials for the project.
- iv. Access the Programmable Search Engine and enable the “Image search” option within the web form where you create or edit your custom search engine. For the “Sites to search” option, select “Search the entire web but emphasize included sites”.

After completing the setup of Google developers account and project, you should receive your developer API key and project CX, CX being Google’s Custom Search Engine ID.

2.2. Build the Feature Dataset

We manually extracted parts of UML diagrams using the built-in macOS screenshot tool, which also helped define the file locations. First, we opened the UML diagram on our screens, with images which were previously downloaded as described above. By pressing Command + P, a custom shortcut, we activated the screenshot tool, changing the cursor to a crosshair. We then clicked and dragged to select the area of the UML diagram we wanted to capture. Upon releasing the mouse button, the screenshot was taken and automatically saved to the desired location. The file locations were clearly indicated as the screenshots were named with timestamps and saved in a predefined folder, ensuring easy retrieval and organization of the extracted parts.

To conduct training on a feature-based dataset, specific parts of the diagrams must be extracted. For UML diagrams, this involves identifying and isolating key modeling concepts such as classes, interfaces, relationships, and other elements defined in the UML specification. Examples of features are listed in Table 3 below and follow the guidance and official naming of modeling elements from [10]. These elements are then converted into feature vectors that can be fed into the neural network for training.

Table 3. UML diagram features introduced in the training dataset.

	Class Diagram	Sequence Diagram	State Diagram
Features	Class	Lifeline	State
	Abstract Class	Actor	Initial State
	Interface	Message	Final State
	Nested Classifier	Synchronous Message	Composite State
	Object	Asynchronous Message	Substate
	Enumeration	Create Message	Entry Point
	Association	Destroy Message	Exit Point
	Operation	Return Message	Transition
	Constraint	Combined Fragment	Trigger
	Multiplicity	Interaction Use	Guard
	Composite		
	Aggregation/Composition	Execution Specification	Action
	Association Navigability	State Invariant	Self-Transition
	Generalization	Continuation	Fork Node
	Dependency	Time Constraint	Join Node
	Required interface	Duration Constraint	Choice Node
			Junction Node
			History State
			Termination

Non-UML diagrams were sourced from the same Google Images Search [14] using a custom Python script. This script leverages the Google Custom Search API to retrieve images that match specific keywords related to diagrams that are not UML, but close to the UML definition. These include Business Process Model and Notation (BPMN), Generic

Modeling Environment (GME), Entity–Relationship Diagrams (ERDs), and Data Flow Diagrams (DFDs).

The non-UML diagram dataset followed these steps:

- i. Enable the Google Custom Search API
- ii. Configure search parameters: The script is configured to search for keywords like “BPMN diagram”, “ERD diagram”, “DFD diagram”, and “GME diagram”.
- iii. Download images: The script downloads the images matching these keywords and saves them in a predefined directory for further processing.

Additionally, to ensure a robust training dataset, manual curation was performed. This included adding samples that resemble UML diagrams but are not strictly UML. These samples help the model to learn subtle differences and avoid misclassification.

2.3. File Processing Required Prior to the Training Phase

After manually extracting parts of the UML diagrams using the built-in macOS screenshot tool, we proceeded to rename all gathered images systematically. Each screenshot, initially saved to the required directory, was renamed using the format <UML_DIAGRAM_NAME>_<NUMBER>. We started by identifying the UML diagram name for consistency. Then, we renamed files in bulk, appending a sequential number to maintain order. For instance, if the UML diagram name was “class”, the files were renamed as “class_1”, “class_2”, and so on. For UML and non-UML features, we named them for example “uml_1”, “uml_2” and “non_uml_1”, “non_uml_2” respectively. This organized approach ensured that the images were easily identifiable and systematically ordered, facilitating efficient retrieval and reference during image processing and training phases.

Given the project used parts of UML diagrams, representing features to train the model, each image was saved at 150×150 px. The images were not directly saved in 150×150 px size, and we used Python’s library pillow or PIL <https://pypi.org/project/pillow/> (accessed on 10 May 2024). Pillow is a widely used Python library for image processing that offers an easy-to-use interface for tasks such as loading, manipulating, and saving images. It supports various image processing functions including resizing, cropping, filtering, and format conversion, making it an indispensable tool for handling and comprehending images.

We created the `resize_and_pad` function, a robust image processing method designed to resize an input image to fit within specified dimensions while maintaining its aspect ratio. Additionally, it adds padding to ensure the final image precisely matches the desired size. By default, the padding color is white, but it can be customized as needed.

The function involves the following key steps:

- i. Image resizing: The function first resizes the input image. This step ensures that the image fits within the specified dimensions while preserving its original aspect ratio.
- ii. Padding addition: Next, padding is added to the resized image, and this extends the canvas to the target size, with the padding area filled in the specified color.
- iii. Saving the result: Finally, the processed image is saved to the designated output.

Adjusting the size parameter allows for control over the image dimensions, and the background color can be set to suit specific requirements. In our case, white was the best since images contain such a background.

All images were in either .png or .jpeg format, with most of them being black and white, and only some being colored as well. For each category, we saved 750 images. The total size of the data for this project was 82.42 MB, with an average of 16.48 MB per category. The split between .png and .jpeg files in each category was not balanced, nor the same across every directory.

2.4. Expanding the Dataset: Data Augmentation

Prior to the actual preparation of data for training, we performed data augmentation to expand the dataset. For the data augmentation process we carried out, we applied several transformations to our dataset, including zoom in/out, colorization, and image translation.

- i. Zoom in/out: We adjusted the scale of each image in the dataset by either zooming in or zooming out. This was achieved by enlarging or reducing the image size by a certain percentage. The zoom levels were randomly selected within a predefined range to ensure that our model could recognize features at different scales and distances.
- ii. Random colorization: To introduce variability in color, we altered the color properties of the images randomly. This included adjusting the hue, saturation, and brightness. This step aimed to train the model to perform well under various lighting conditions and color settings.
- iii. Image translation: We also randomly shifted the images along the x and y axes. This shift involved moving the image a few pixels horizontally or vertically. This technique helped make the model robust to positional changes, ensuring it could detect and classify objects correctly even if they were not centrally aligned.

To achieve data augmentation, we exclusively used OpenCV (cv2). OpenCV (Open-Source Computer Vision Library) [15] is a widely used open-source library for real-time computer vision and image processing. Developed initially by Intel, it provides extensive tools for tasks such as object detection, face recognition, and image segmentation. Supporting multiple programming languages, including Python and C++, OpenCV is renowned for its versatility and efficiency, making it a popular choice for developers and researchers in the field of visual data analysis.

With cv2, we implemented random zoom in and zoom out transformations by resizing the images within a specified range to introduce variability in scale. Additionally, we applied image shifting, translating the images horizontally and vertically by a random number of pixels to ensure positional diversity. These transformations enriched the dataset, enhancing the model's ability to generalize across different scales and positions, thereby improving its robustness and performance. Given that the input was small in size now, we augmented the dataset so each category was double the amount before this step, meaning 1500 images for each category.

2.5. Image Processing

The images were processed in the form of a pixel matrix. For this stage, we used the Jupyter Notebook development platform because it allowed us to split the program into smaller parts that could be tested independently. Each sequence in this stage could be verified separately, and the Jupyter Notebook provided quick verification methods for these code snippets, both for numerical values and graphical representations when needed. Our image processing required increased attention because if the data were processed incorrectly, there was a risk of conducting training that would ultimately lead to no results. Therefore, many checks were needed throughout this phase to observe at each step if the proposed goal had been achieved.

Proper processing means that the data content in the form of a pixel matrix can still represent the essential features of the object we want to identify. Processing is considered unsuccessful if the image represented as a pixel matrix is altered and the object in the image is no longer understandable, even by a human operator.

At this stage, we first loaded the images with a special reading function to ensure that they were read correctly and could be further processed. For verification, we displayed an image from each category. In the representation below, it can be seen that the image dimensions are specified, namely 150×150 pixels, and the pictures may contain colors, thus meeting the requirements we highlighted in the previous phase.

For artificial intelligence systems, images have no relevance in the form we perceive them. Therefore, they need to be processed into a three-rank tensor. The rank comes from

the fact that each pixel can be represented as a triplet (r, g, b) because the images in the database are colored. The three parameters represent the red (r) light intensity, the green (g) light intensity, and the blue (b) light intensity, which form the content of a pixel in the image.

Thus, we had three matrices whose size was determined by the image resolution, so each matrix had 150 rows and 150 columns. In the end, all the collected images from the database went through this processing method, resulting in a fourth-rank tensor, with the number of images representing the parameter extending the tensor's rank.

For processing, we used the NumPy library in Python, which is widely used for scientific calculations. The necessary parameters for this step were the image dimensions, the number of channels, and image encoding. The dimensions were well known at this point, the number of channels was three because the image could be colored, and for encoding, we found that there were two commonly used options. One encoding was float32, which was used when the image was grayscale, and the other was called uint8 (unsigned 8-bit integers), used for colored pictures. When transitioning from the image to the pixel matrix, the content was of type uint8.

After transitioning from the image to the pixel matrix, it had to be verified that no important features from the image content were lost. For this verification, we calculated two error measurement functions, namely the mean error and the standard deviation, to evaluate how much information was lost from the original image after processing.

The mean error is the fundamental measure in statistics. In simpler terms, it measures how far the image obtained after processing is from the original. A small mean error indicates small losses in the image content.

As stated in [16], the standard deviation is a measure of variation or diversity from statistics. When it comes to image processing, it shows how much variation or “dispersion” there is on average. A small standard deviation indicates that the points are close to the mean value, while a large one shows that the points cover a wider range of values. More concretely, for our application, a small standard deviation indicates that important features in the image have been preserved.

For the proposed application, we imposed a mean error that should not exceed zero and a standard deviation that should not exceed 0.5. In Table 4 it can be seen that the criteria set at the beginning were met, so we can consider that the images were processed correctly, and no important content was lost from the images.

$$MAE = \frac{\sum_{i=1}^n |y_i - x_i|}{n} \quad (1)$$

where MAE is the mean absolute error, y_i is the prediction, x_i is the true value, and n is the total number of data points.

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}} \quad (2)$$

where σ is the population standard deviation, N is the size of the population, x_i is each value from the population, and μ is the population mean.

Table 4. Mean and standard errors for every processed class.

	class.pickle	non_uml.pickle	sequence.pickle	state.pickle	uml.pickle
Mean	−0.0044898701	−0.082445047	−0.099869774	−0.042088231	−0.1001582262
Std	0.3404123	0.2871922	0.29997926	0.30228693	0.29298602

In the previous figure, it can also be seen that each category is followed by a “.pickle”, representing a file extension where we saved the values after the processing stage because they would be used later in a new training script. Pickle is a module offered by Python for data storage and it has the advantage of saving data in the file as a string; when reading, it provides a set of tools to interpret the content in the desired form. Considering that we

are still dealing with images, we could make a final check by trying to interpret the pixel matrices obtained as images.

We listed the image samples used before and after processing for the manual testing of any alteration. Given that the majority of the dataset is black and white, there was no significant loss that a human operator could capture. In comparison with the initial images, processing and re-displaying them led to a slight blur and color alteration that could be captured. However, no loss was expected to impact the training, since all important features were kept.

At this point, we could say that the image processing stage was successfully completed, and we could move on to the next stage, which truly represents the artificial intelligence system, namely the preparation for training.

Loading and processing training and test data for image-based machine learning tasks often requires the use of efficient libraries like `os` and `cv2` in Python. The `os` module is indispensable for navigating the filesystem to gather image paths, while `cv2` (OpenCV) is essential for reading and processing the images. To start, the `os` module can be used to traverse the directory structure, collecting the paths to all images within specified folders for training and testing datasets. This ensures that we have a comprehensive list of images to work with for further processing.

Once the paths are collected, we use `cv2` to read each image. A critical step in this process is ensuring that all images conform to a specified resolution. In real-world datasets, images often come in various sizes and resolutions. To standardize these images, we can use `cv2.resize()` to resize them to a required resolution, say 150×150 pixels. This is particularly important because most neural networks require input images of a consistent size. The resizing process involves specifying the desired width and height, and choosing an interpolation method, such as `cv2.INTER_LINEAR` or `cv2.INTER_AREA`, to maintain image quality during resizing. In our case, we performed this step earlier in the process.

2.6. Labels Processing

After image processing, corresponding labels should be handled with equal care, ensuring they are saved as `int32` to match the expected input types for most machine learning frameworks. This can be achieved using `label.astype('int32')` to ensure consistency and avoid potential type mismatch issues during model training.

To save these processed images and labels, numpy arrays can be used. The images can be stored in arrays with a shape that reflects the number of images and their dimensions, for example `(num_images, 150, 150, 3)` for RGB images. These arrays can then be saved using numpy's `np.save` function, which preserves the data types and ensures that the images and labels are stored in a format that is quick to load during training. This entire workflow ensures that the data fed into machine learning models is of consistent quality and format, thereby facilitating effective and efficient training processes.

The number of examples in both the training and testing datasets was determined by examining the shape of the label arrays for each set. This allowed for an accurate count of how many samples were present in each dataset. Specifically, the training dataset comprised a certain number of examples, while the testing dataset contained a different number. This count is critical for understanding the scale of the datasets used in the experiments.

Additionally, the size of each image was confirmed to be consistent across the dataset. The image size, denoted as `IMAGE_SIZE`, was uniform for all images, ensuring that each image conformed to the required dimensions for input into the neural network. This uniformity in image size is essential for the preprocessing pipeline and subsequent model training, as neural networks typically require inputs of a consistent shape and size. By reporting the number of training and testing examples, as well as the uniform image size, the research provides clear and detailed information on the dataset's characteristics.

To visually display the distribution of images across different categories in the training dataset, a pie chart was created, as shown in Figure 1. This chart illustrates the proportion of images belonging to each category (i.e., class, sequence, state machine, UML diagrams

or non-UML, images that do not contain UML diagrams), providing a clear overview of the dataset composition. The chart includes labels for each category and displays the percentage of the total dataset that each category represents, formatted to one decimal place. Ensuring the pie chart is a perfect circle enhances its readability and esthetic appeal. The chart is displayed in Figure 1, clearly communicating its purpose. This visualization effectively highlights the relative proportions of images in each category within the training dataset, offering valuable insights into the dataset's structure.



Figure 1. Dataset distribution per category: (a) UML diagram types (b) UML vs. non-UML.

2.7. Neural Network Training

As next steps, we scaled the data. Typically, the values in an image dataset are pixel intensity values, which usually range from 0 to 255 for each color channel (red, green, blue) in an 8-bit image. By dividing these values by 255.0, the code normalizes the pixel values to a range between 0 and 1.

Neural networks work better when the input values are in a consistent range, typically between 0 and 1 or -1 and 1 . Normalized data helps in faster convergence during gradient descent. It helps in stabilizing and speeding up the training process because it ensures that the weights and biases of the neural network adjust at similar scales.

Large input values can lead to numerical instability, making it harder for the model to converge or leading to very high losses initially. By scaling the values down, it prevents potential issues related to large input values.

Many machine learning algorithms, including neural networks, perform better when the input features are normalized. This is because normalization helps in maintaining the relationship between different pixel values and ensures that the training process gives equal importance to all features.

Consider an example of an image with the following pixel values:

[[0, 0, 0]	[[0, 255, 0]
[255, 255, 255]	[0, 0, 255]
[128, 128, 128]	[255, 0, 0]

The pixel values will be scaled to:

[[0.0, 0.0, 0.0]	[[0.0, 1.0, 0.0]
[1.0, 1.0, 1.0]	[0.0, 0.0, 1.0]
[0.5, 0.5, 0.5]	[1.0, 0.0, 0.0]

By scaling the data in this manner, one ensures that the input values are in a range that is more suitable for training a neural network, leading to better performance and more stable training.

Before defining the neural network, it is good practice to display a random image from the dataset to make a final check that the data are in good condition to be used during the training phase. This step helps verify that the images are correctly preprocessed and normalized, ensuring they are suitable for input into the neural network.

Using a Multi-Input Multi-Layer Perceptron (MLP) as a neural network for a feature-based dataset is an approach that leverages the power of deep learning to handle complex, multi-dimensional data. Given that the project focuses on feature extraction from UML diagrams, an MLP is ideal for processing these predefined features rather than relying on raw image data. This architecture is computationally less intensive compared to more complex models like convolutional networks, making it well-suited for projects with smaller datasets and structured features, ensuring faster training times. Additionally, the multi-layer structure of the MLP enables the model to capture non-linear relationships between diagram features, which is crucial for accurate classification.

For feature-based datasets, where the input consists of various types of features (e.g., numerical, categorical, text), a Multi-Input MLP can be particularly effective. In such networks, different types of input features are often processed through separate sub-networks (sub-MLPs) tailored to each feature type. These sub-networks extract relevant information from each feature type before merging their outputs in a concatenation layer, which is subsequently fed into additional hidden layers for further processing. This architecture allows the model to handle heterogeneous data efficiently and learn from the combined information in a cohesive manner [17,18].

The training process of a Multi-Input MLP involves adjusting the weights of the neurons using backpropagation and gradient descent. The backpropagation algorithm calculates the gradient of the loss function with respect to each weight by the chain rule, allowing the network to learn the optimal weights that minimize the prediction error on the training data. Regularization techniques such as dropout and batch normalization can be employed to prevent overfitting, ensuring that the model generalizes well to unseen data [19,20].

The effectiveness of Multi-Input MLPs in handling feature-based datasets has been demonstrated in various domains. For example, in healthcare, these networks have been used to integrate patient demographic information, clinical measurements, and genomic data to predict disease outcomes [21]. In finance, Multi-Input MLPs have been utilized to combine market indicators, economic factors, and textual news data to forecast stock prices [22]. These applications highlight the versatility and robustness of Multi-Input MLPs in processing and learning from complex, multi-faceted datasets.

In summary, employing a Multi-Input Multi-Layer Perceptron for feature-based datasets offers a powerful approach to capturing the complex interactions within heterogeneous data. By leveraging the architecture of MLPs and the flexibility to handle multiple input types, this method provides significant advantages in terms of accuracy and generalization across a wide range of applications [23,24].

To define the neural network, we used Keras. Keras is a powerful, high-level neural network API, written in Python and capable of running on top of TensorFlow. It is designed to enable fast experimentation with deep neural networks, allowing users to build and train models efficiently. Keras provides a user-friendly interface that simplifies the process of building complex neural network architectures. It supports both convolutional and recurrent networks, as well as combinations of the two, making it a versatile tool for various deep learning tasks [25].

To define a multi-input neural network in Keras, each input is specified with its respective shape. For example, images of size 150×150 pixels are represented as separate input layers. Each input undergoes a flattening process to convert the 2D matrices into 1D arrays. Dense layers are then applied to each flattened input, allowing the network to learn

representations from each input independently. These dense layers are concatenated to merge the features learned from each input, followed by additional dense layers to process the combined features further. This approach enables the network to learn from multiple inputs simultaneously, making it suitable for complex, multi-feature datasets [18].

In the neural network, we integrated early stopping. Early stopping is a technique used to prevent overfitting in neural networks by monitoring the model's performance on a validation set during training. It stops training when the validation loss stops improving, indicating that the model has learned as much as it can from the training data. The patience parameter specifies the number of epochs to wait for an improvement before stopping. The `restore_best_weights` parameter ensures that the model weights are set to the best-performing epoch, providing the best possible model after training. This technique is crucial for achieving optimal generalization on unseen data [26].

Setting a high number of epochs, such as 100, allows the model to learn sufficiently from the data, ensuring that it captures the underlying patterns. However, this must be balanced with the risk of overfitting. Early stopping can mitigate this risk by halting training when performance ceases to improve. Training for more epochs provides a comprehensive understanding of the training dynamics and helps determine the optimal stopping point based on validation performance [18].

Choosing a batch size of 64 balances the trade-off between computational efficiency and the stability of gradient updates. Larger batch sizes, such as 64, enable more stable and accurate gradient estimates, leading to smoother convergence. They also leverage modern hardware capabilities, such as GPUs, more effectively, speeding up the training process. However, very large batch sizes may require more memory and might not generalize as well as smaller batches. Therefore, a batch size of 64 is often a practical choice for training deep neural networks [27].

Validation split is a technique used to assess the generalization performance of a neural network. During training, a portion of the data, typically 10–20%, is set aside as a validation set. This set is not used for training but for evaluating the model's performance after each epoch. By monitoring the validation metrics, such as loss and accuracy, we can detect overfitting and underfitting. The validation split helps in tuning hyperparameters and implementing early stopping, ensuring that the model performs well on unseen data [18].

Figure 2 displays the architecture defined in regard to MLP neural networks. The last step before training is the compilation of the model. The optimizer selected for the model was the Adaptive Moment Estimation (Adam) optimizer [28]. Adam is an advanced optimization algorithm that combines the advantages of two other popular methods: Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). Adam utilizes first-order gradients to update the learning parameters and adapts the learning rate for each parameter based on the estimated moments of the gradients. This results in faster convergence and improved performance on complex, high-dimensional data. Adam's adaptability and efficiency make it a highly effective choice for a wide range of machine learning tasks, especially those involving large datasets and complex model architectures, as highlighted in [29].

The loss function employed is sparse categorical crossentropy. In classification problems where the target variable represents discrete categories, crossentropy is a standard choice for quantifying the difference between the predicted probability distribution and the true distribution. Sparse categorical crossentropy is particularly suited for problems where the target data are represented as integers rather than one-hot encoded vectors. This approach simplifies the data preprocessing steps and reduces computational overhead, while still effectively guiding the model to minimize the prediction error for multi-class classification tasks.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 150, 150, 1)	0	–
input_2 (InputLayer)	(None, 150, 150, 1)	0	–
input_3 (InputLayer)	(None, 150, 150, 1)	0	–
input_4 (InputLayer)	(None, 150, 150, 1)	0	–
flatten_1 (Flatten)	(None, 22500)	0	input_1[0][0]
flatten_2 (Flatten)	(None, 22500)	0	input_2[0][0]
flatten_3 (Flatten)	(None, 22500)	0	input_3[0][0]
flatten_4 (Flatten)	(None, 22500)	0	input_4[0][0]
dense (Dense)	(None, 128)	2,880,128	flatten_1[0][0]
dense_1 (Dense)	(None, 128)	2,880,128	flatten_1[0][0]
dense_2 (Dense)	(None, 128)	2,880,128	flatten_2[0][0]
dense_3 (Dense)	(None, 128)	2,880,128	flatten_3[0][0]
dense_4 (Dense)	(None, 128)	2,880,128	flatten_4[0][0]
concatenate (Concatenate)	(None, 640)	0	dense[0][0], dense_1[0][0], dense_2[0][0], dense_3[0][0], dense_4[0][0]
dense_5 (Dense)	(None, 64)	41,024	concatenate[0][0]
dense_6 (Dense)	(None, 5)	325	dense_5[0][0]

Total params: 14,441,989 (55.09 MB)
 Trainable params: 14,441,989 (55.09 MB)
 Non-trainable params: 0 (0.00 B)

Figure 2. Model’s defined architecture to be used during the training phase.

To evaluate the performance of the model during training and validation, the metric chosen was accuracy. Accuracy is a straightforward and intuitive measure that calculates the proportion of correctly predicted instances out of the total instances. By tracking accuracy, researchers can gain insights into a model’s performance in correctly classifying the input data. This metric is especially useful in balanced datasets where all classes are represented equally, providing a clear indication of the model’s classification capabilities.

When running ‘model.fit’ in Keras, you will observe detailed progress information for each epoch in the console. This includes the current epoch number out of the total epochs, the training loss and accuracy metrics for each batch and epoch, and the validation loss and accuracy if a validation split is used. Additionally, you see the time taken per epoch. If early stopping is enabled, a message will indicate when training halts early due to no further improvement in validation performance. This output is crucial for monitoring the training process and adjusting hyperparameters as needed.

3. Results

Section 3.1 focuses on the model’s performance from the metrics that were tracked during the training phase. It also contains a comparison with previous research. Section 3.2 lists the recommended metrics to be evaluated in regard to performance.

3.1. Model Evaluation and Testing

Once training was conducted, we plotted the metrics to observe the model’s performance. A steadily decreasing training loss and increasing training accuracy indicate that the model was effectively learning the patterns present in the training data. If the validation loss also decreased and validation accuracy increased in a similar manner, it would indicate that the model is generalizing well to unseen data, which is a sign of a robust and effective model.

However, if the training loss continues to decrease while the validation loss starts to increase, this is a clear indication of overfitting. Overfitting occurs when the model learns the noise and details in the training data to such an extent that it negatively impacts its

performance on new data. This is often observed when the training accuracy is significantly higher than the validation accuracy. The model performs well on the training set but fails to generalize to the validation set.

Conversely, if both the training and validation losses remain high and the accuracies are low, the model is likely underfitting. Underfitting occurs when the model is too simple to capture the underlying patterns in the data. This can happen if the model lacks the necessary complexity, such as insufficient layers or neurons, or if it is not trained for enough epochs. In this scenario, the model fails to perform well on both the training and validation sets.

The accuracy and loss plots are also crucial for determining model convergence. Model convergence is indicated when the loss values stabilize and no longer improve significantly over successive epochs. This helps in deciding when to stop training. If the plots show a flat trend in loss and accuracy, it suggests that the model has reached its optimal performance given the current training setup. This information is useful for implementing early stopping to avoid unnecessary training time and computational resources.

Figure 3 displays how the model performed during training. The analysis can be either achieved by (a) accuracy or (b) loss. By looking at the accuracy, we can see that the model performs better on training data, which is expected since if data were used in the training process, the accuracy ends with a high value, which may indicate overfitting, but the validation accuracy is high, which indicates the model performs well on unseen data. There are minor spikes in the training phase, and they indicate the model was not well structured. What matters in these cases is that the training does not stop right at that moment, since the model may not be able to perform well. However, the model adjusted by the end and finished with high accuracy and reduced loss.

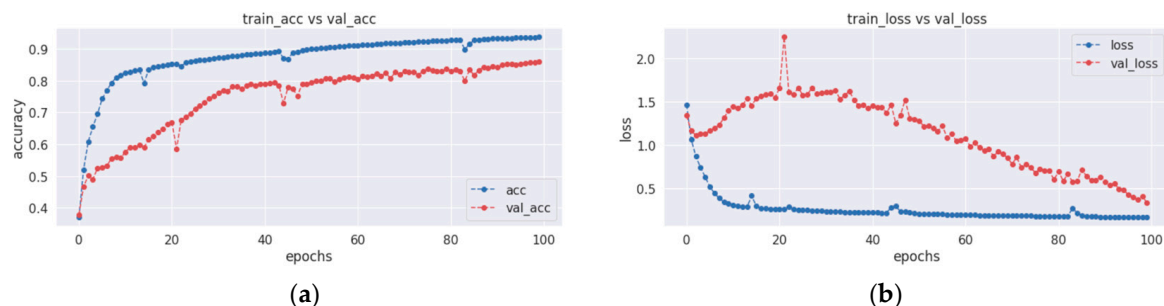


Figure 3. Training and validation: (a) accuracy (left) and (b) loss (right) from last execution.

Saving a trained model for later use is an essential step in the machine learning workflow. In Keras, this can be achieved by saving the model in the .h5 format using the `model.save('model.h5')` method. The .h5 file format is an HDF5 (Hierarchical Data Format version 5) file, which is a versatile data model that can represent complex data objects and a variety of metadata. HDF5 is widely used for storing large amounts of data and is particularly suitable for handling large neural network models. It allows for efficient storage and retrieval of the complete model architecture, weights, and optimizer state, ensuring that the model can be reloaded and used later without any loss of information or functionality.

The process of saving a model involves serializing the architecture, weights, and optimizer information into a single .h5 file (see model details in Table 5). This file can then be loaded later to reconstruct the exact state of the model at the time of saving. This is particularly useful for deploying models in production environments, sharing models with others, or resuming training at a later point. The ability to save and load models in this way ensures that the training process is reproducible and that models can be consistently utilized across different platforms and systems. By using the .h5 format, users can efficiently manage their trained models and ensure that they are ready for deployment or further analysis.

The project progressed through multiple phases to achieve a robust image classification system. Initially, data collection and preprocessing were conducted to ensure high-quality inputs for the model. The next phase involved designing and training a neural network using the Adam optimizer and sparse categorical crossentropy loss function, with accuracy as the evaluation metric.

After thorough evaluation and optimization, we developed a custom prediction function. This function takes a tested image and utilizes the trained model to classify the image accurately. The custom predict function streamlines the deployment process, ensuring that the model can be effectively used for real-time image classification tasks. This approach was iteratively refined based on testing and validation results, ensuring high performance and reliability in the classification system.

Table 5. Results of current research against previous experiment.

	MLP Network Previous Experiment	MLP Network Current Experiment
UML diagram categories	class	class, sequence, state machine
Feature dataset size	2000	1500
Class diagram correct classification	88%	92%
Average accuracy score	89%	90%

During the entirety of the research, we kept 25 samples to compare results across experiments. The samples consisted of five class diagrams, five state diagrams, five sequence diagrams, five other UML diagrams, and five images unrelated to UML diagrams. Each UML sample consisted of an entire image in the .pdf or .jpg format with no constraints on a fixed size. For the current experiment, we obtained the following results: 18 images correctly classified as true positives, 2 images as false positives, 3 false negatives, and 2 true negatives. All the edge cases that we tracked during research were correctly classified.

In order to help the model generalize well, we considered all features that could be a part of each UML category, especially the diagram types we focused on classifying. In order to support the correct classification, we focused on specific features. One edge case we focused on was a Generic Modeling Environment against UML class diagrams, where many features are similar. However, there are features that indicate that the difference as well as connection between components may highlight why a certain diagram falls into a specific category. For the non-UML category, we focused on bringing these features in so the model could capture these details and make the right classification.

To further evaluate the robustness of our model, we tested its performance under varying conditions. The model was tested on images with added noise to simulate low-quality images containing UML diagrams. The accuracy decreased slightly by 1.2%, demonstrating that the model is robust to image quality variations. We introduced small random rotations and scaling variations to the input images. This could map to a scenario where the image comes from a scanned paper where the image is not always straight. The model retained an accuracy of 87.8%, showing its ability to handle variations in input data.

The current model is aimed to be used for educational purposes. First, by looking at features, the model can support identifying syntax errors what would mark the diagram to no longer be part of the intended category. Additionally, it could be integrated to identify similar diagrams among students to support faster evaluation and ensure integrity and ethical standards.

3.2. Performance Evaluation

Using the core 25 samples, we obtained a precision of 0.9, indicating that 90% of the instances it identified as positive were indeed correct. The recall was approximately 0.857, reflecting the model's ability to correctly identify about 85.7% of all actual positive cases. The F1 score, which balances precision and recall, is approximately 0.877, showing that the model performs well in terms of both accuracy and recall. The overall accuracy of the

model is 0.8, meaning that 80% of all predictions made by the model, both positive and negative, were correct. However, the specificity is 0.5, indicating that the model correctly identified 50% of the actual negative cases. This suggests that while the model is effective at identifying positive cases, there is room for improvement in distinguishing negative cases.

The validation of the classification model is demonstrated through the detailed results for various configurations of Single-Input Single-Output (SISO) and Multiple-Input Multiple-Output (MIMO) models. For the SISO 1 configuration, the model achieved a precision of 0.82, a recall of 0.73, and an F1 score of 0.77. The MIMO 1 configuration showed improved performance with a precision of 0.93, a recall of 0.75, and an F1 score of 0.83. Further enhancements were observed in the SISO 2 configuration, where both precision and recall were 0.89, leading to an F1 score of 0.89. The highest performance was recorded with the MIMO 2 configuration, which achieved a precision of 1.00, a recall of 0.85, and an F1 score of 0.91.

The plots illustrate the accuracy and loss of a model during training and validation phases over 100 epochs. The left plot represents the accuracy, with the blue line indicating training accuracy and the red line indicating validation accuracy. The training accuracy starts at approximately 0.45 and steadily increases, reaching around 0.9 by the end of the training period. Similarly, the validation accuracy begins at around 0.45 and rises to about 0.85 by the final epoch. This steady improvement and convergence suggest that the model was learning effectively.

The right plot shows the loss, where the blue line represents training loss and the red line represents validation loss. Initially, the training loss is high, around 2.0, but it decreases to below 0.5 by the end of the training period. The validation loss follows a similar pattern, starting high at around 2.0, experiencing a slight increase, and then decreasing to stabilize around 0.6. The decrease in both training and validation losses indicates successful training.

The convergence of the training and validation accuracies, coupled with the decrease in losses, suggests that the model is not overfitting. The slight gap between training and validation performance is typical and expected in well-trained models. The model achieves high accuracy and low loss on both training and validation sets, aligning with the desired training outcomes, as highlighted in [30]. This paper discusses the ResNet architecture, which achieves high accuracy on image recognition tasks and includes an analysis of training and validation accuracy as well as loss curves.

Throughout both plots, we observe occasional spikes, particularly noticeable in the validation loss. These spikes are common in the training of neural networks and can occur due to several reasons:

Mini-batch Variability: During training, the model updates its parameters based on mini-batches of the training data. Some mini-batches may contain more challenging or noisy examples, causing temporary increases in loss.

Learning Rate Adjustments: If the learning rate is adaptive or scheduled to change during training, sudden changes in the learning rate can cause temporary spikes in loss as the model adjusts to the new rate.

Model Exploration: At certain points, the model might be exploring different regions of the parameter space, especially early in training. This exploration can lead to temporary instability in the loss and accuracy.

Overfitting on Specific Batches: In some cases, the model might overfit to specific mini-batches during training, leading to sudden decreases in validation accuracy or increases in validation loss when evaluated on the entire validation set.

These spikes, while they can appear alarming, are typically part of the normal training dynamics. As long as the overall trend in the loss is downward and the accuracy is upward, the training process is generally progressing well.

The model depicted in the provided plots appears to be performing well, with high and converging accuracies and low and decreasing losses. This indicates effective learning without significant overfitting, aligning with the performance characteristics discussed in

the referenced research paper. The image in Figure 4 was classified as a UML diagram with a 92% accuracy score and as a class diagram with a 93% accuracy score.

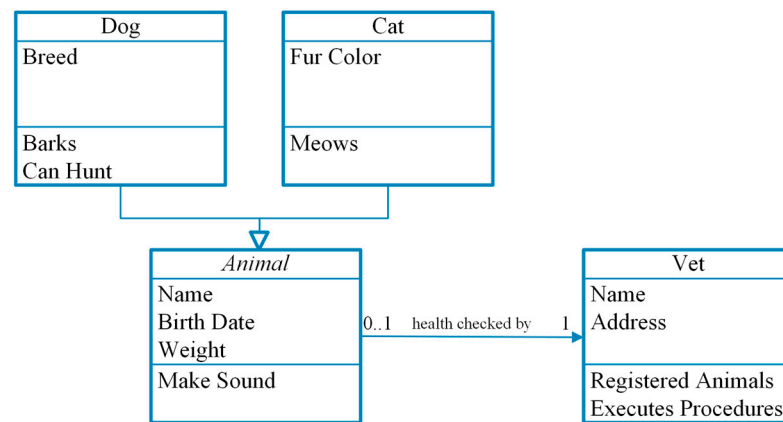


Figure 4. Example of class diagram used for testing.

The model's capabilities have been further enhanced to include the classification of state machine diagrams, which are used to depict the states of an object and the transitions between these states in response to events. By employing refined feature extraction techniques, the model identifies critical components such as states, transitions, events, and actions within state machine diagrams. This enables the model to accurately distinguish state machine diagrams from other types of UML diagrams. The image in Figure 5 has been classified as a UML diagram with a 91% accuracy score and as a state machine diagram with a 93% accuracy score.

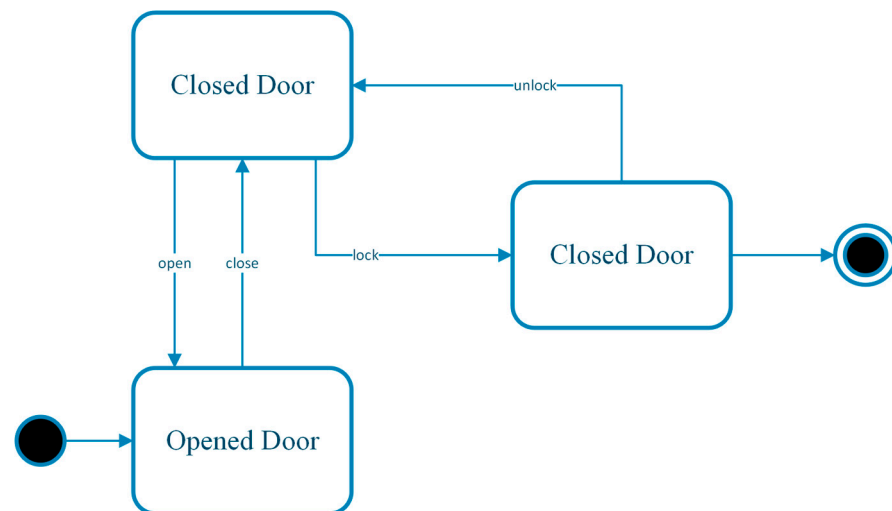


Figure 5. Example of state machine diagram used for testing.

The enhanced model extends its classification capabilities to sequence diagrams, which represent the interactions among objects in a specific sequence of time. Utilizing advanced feature extraction techniques, the model captures the unique temporal and structural characteristics inherent in sequence diagrams. This includes identifying key elements such as lifelines, messages, and activation bars. By leveraging these features, the model effectively differentiates between sequence diagrams and other UML diagrams.

In comparative testing, the model demonstrated higher accuracy in sequence diagram classification than for previously utilized models. This improvement is attributed to the integration of a more sophisticated neural network architecture and enhanced training methodologies, including comprehensive data augmentation and robust feature selection.

The current model's accuracy surpasses that of earlier versions, confirming its capability in handling complex diagram classifications. The image in Figure 6 has been classified with a 92% accuracy score as a UML diagram and with a 95% accuracy score as a sequence diagram.

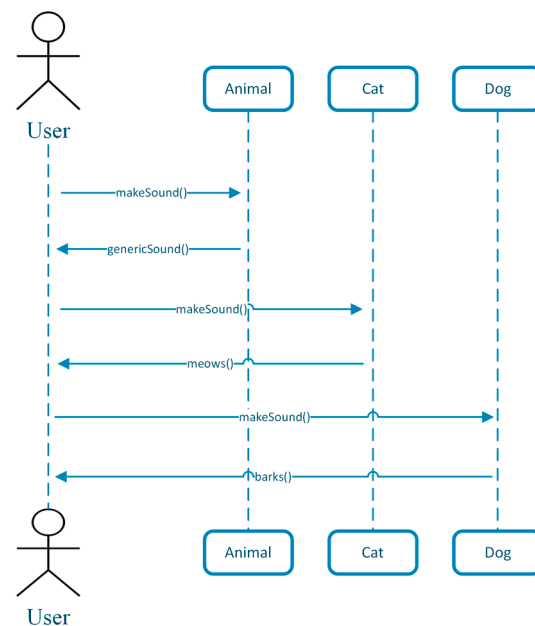


Figure 6. Example of sequence diagram used for testing.

Manual testing of the model further validated its accuracy and robustness, as it consistently achieved correct classifications in most cases. Human testers meticulously reviewed and assessed the model's predictions against a diverse set of data, including standard and edge cases. This comprehensive evaluation ensured that the model's performance was reliable and consistent. The manual testing process provided an additional layer of assurance, highlighting the model's capability to generalize well across various inputs and to maintain high classification accuracy, further reinforcing its superiority over previous models and automated tools.

The current model achieves higher classification accuracy than previous models in the same research by leveraging several advanced techniques. It likely benefits from enhanced data preprocessing and augmentation methods, which improve data quality and variety. The use of a more sophisticated architecture, such as deeper networks or those with residual connections, allows the model to capture complex patterns more effectively. Additionally, optimized training techniques, including better weight initialization and effective regularization methods like dropout and batch normalization, contribute to faster convergence and improved generalization. These combined advancements result in superior performance compared to earlier models.

4. Discussion

These results illustrate that the MIMO models generally outperform the SISO models, particularly in terms of precision and F1 score. Notably, the MIMO 2 configuration's perfect precision and high F1 score indicate a robust balance between precision and recall, confirming the model's efficacy in accurately classifying UML diagrams.

While the initial MLP network, focused solely on class diagram classification, achieved an F1 score of 0.92, it is important to note that the new MLP model, despite achieving a slightly lower F1 score of 0.91, is still highly effective. This is particularly commendable given the expanded set of categories now included in the classification task (see Table 6). The addition of state machine and sequence diagrams increased the complexity of the classification problem, yet the model has maintained a high level of performance. This underscores the robustness and adaptability of the new model in handling a more diverse set

of UML diagrams, ensuring continued accuracy and reliability across various classification tasks within software engineering.

Table 6. Personal research comparisons.

	MIMO and SISO 1 [31]	MIMO and SISO 2 [31]	MLP 1	MLP 2
UML diagram categories	class, sequence, state	class, sequence, state	class	class, sequence, state
Dataset type	Full image (diagram)	Full image (diagram)	Feature of diagram	Feature of diagram
Feature dataset size	250	500	2000	1500
Can handle edge cases	No	Yes	Yes	Yes
F1 score	0.83 and 0.77	0.91 and 0.89	0.9	0.87

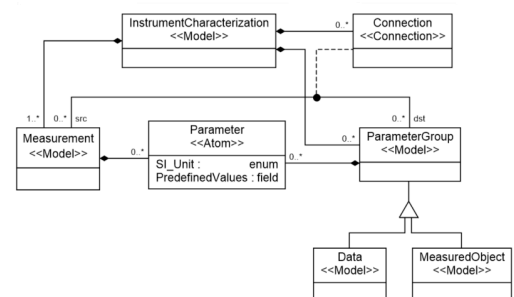
By providing these detailed metrics, we ensure a comprehensive validation of the model's performance, addressing the gap in the initial report and substantiating the claims of improved accuracy and robustness.

The model's ability to handle edge cases was rigorously tested using advanced tools such as ChatGPT 4 Pro version and GitHub Copilot 1.84, which previously struggled with correctly classifying these challenging scenarios, as displayed in Figures 7 and 8. The model excels in managing edge cases due to its well-prepared dataset, which includes advanced feature extraction layers and robust error-handling mechanisms. These features enable the model to capture nuanced patterns and subtle variations in data that are often missed by other models. Additionally, the model employs an extensive training dataset that includes a diverse range of examples, ensuring it is well-equipped to generalize and perform accurately across atypical and complex inputs.

Despite these advancements, there are limitations that need to be addressed. The model occasionally fails to classify edge cases accurately, indicating potential gaps in the dataset. These failures suggest that the dataset should be revised to include more specific features or exclude irrelevant ones, ensuring a more comprehensive representation of all possible scenarios.

To understand the performance of our feature-based neural network model, we compared it with existing approaches used for UML diagram classification. These approaches include both traditional machine learning models and deep learning techniques, as reported in previous studies.

Our model achieved an average accuracy of 90% and an F1 score of 0.87, surpassing traditional models such as Support Vector Machines (SVM) and Random Forests, which typically report accuracies around 70–80% for UML diagram classification.



what is in this image?



The image appears to be a UML (Unified Modeling Language) diagram, specifically an object diagram or a class diagram, which is used to show the structure of a system by depicting its classes, their attributes, methods, and the relationships among objects.

Figure 7. Non-UML diagram analysis using ChatGPT.

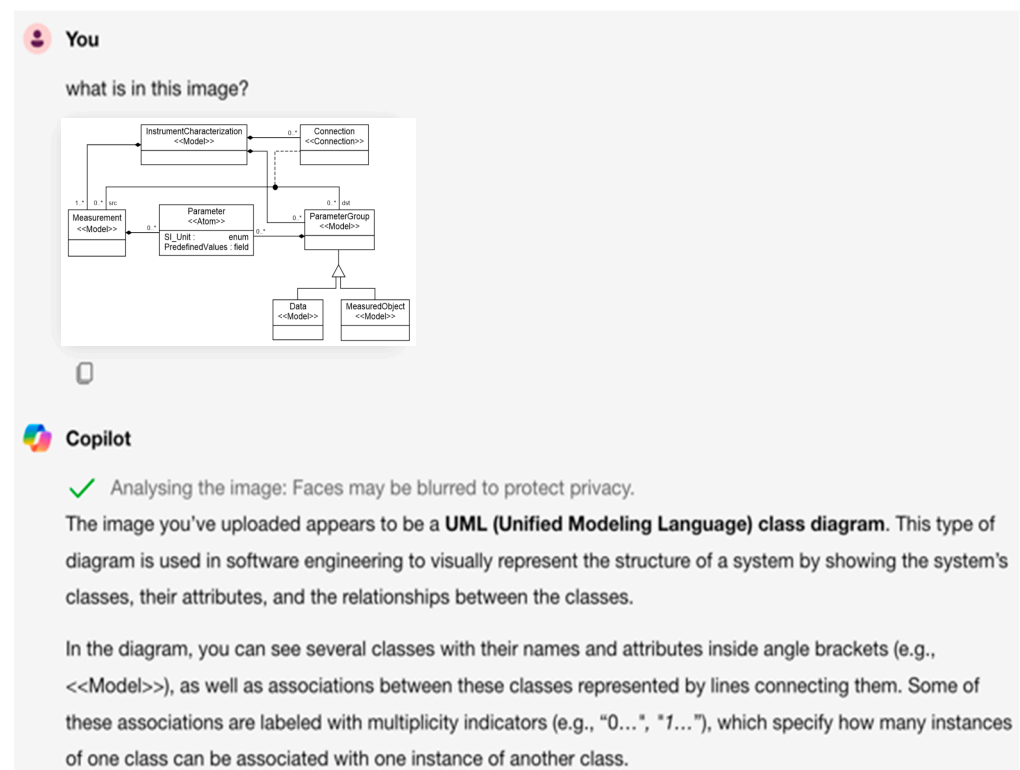


Figure 8. Non-UML diagram analysis using Copilot.

Our model demonstrates superior performance on a wider range of UML diagram types, including class, sequence, and state machine diagrams, compared to [1], who used a CNN-based approach and achieved an accuracy of 88% on class diagrams. Additionally, our model achieves higher precision and recall across multiple diagram types in contrast to [2].

Unlike previous work that primarily focused on class diagrams, the new model effectively handles multiple UML types (class, sequence, state machine), enhancing its applicability in diverse educational and software engineering contexts. By breaking down UML diagrams into their feature components, the model can achieve greater precision in distinguishing between diagram elements.

While the model exhibits strong performance, there are limitations that must be acknowledged. The model's performance relies on the quality and accuracy of the feature extraction process. Any errors in extracting features from UML diagrams could affect classification results. It is important that the data are well understood prior to feature extraction so the specific feature that highlights the diagram type straightaway is heavily present in the dataset. Ideally, each feature would be equally found in the category it belongs to, but some features can be found across multiple diagram types, and then the specific components that belong to only one category are the ones that support the model's ability to generalize.

Outlined below are key recommendations derived from the study's findings, aimed at improving the application of feature-based neural networks in UML diagram classification:

1. **Use of Feature-Based Datasets for Small Data Scenarios:** feature-based datasets reduce the dimensionality and computational cost, allowing neural networks to learn faster and more effectively. Therefore, for projects where data collection is challenging or expensive (e.g., specialized UML diagrams), it is recommended to adopt a feature-based approach, similarly highlighted in [5].
2. **Use of Feature-Based Datasets for Faster Training and Deployment:** as the datasets are inherently smaller, this makes the approach ideal for rapid prototyping and situations where computational resources are limited. Thus, software engineering education

and agile development cycles can benefit from these models to iterate and deploy solutions quickly.

3. **Standardizing UML Diagram Feature Extraction for Consistency:** To support consistent model performance across diverse UML diagram sources, a standardized feature extraction protocol should be developed and followed. This ensures that the models receive well-defined input features, reducing errors during classification and making model training and deployment more reliable across educational and industrial settings. It is recommended to use the official documentation [10] to ensure all research follows the same reference.

5. Conclusions

The performance of the current model was a result of the use of a feature-based dataset combined with the right neural network architecture for such data. This careful feature selection, along with a sophisticated neural network design, significantly enhanced the model's capability to correctly classify even the most challenging scenarios, including edge cases where other models and tools like ChatGPT Pro and GitHub Copilot struggled.

Our study clearly demonstrates the advantages of using a feature-based dataset combined with appropriate neural network architectures for UML diagram classification. The proposed model achieved an average accuracy of 90%, with a precision of 0.9 and an F1 score of 0.877. These results underscore the model's ability to generalize well and deliver superior performance.

Manual testing further validated the model's robustness and high accuracy. Human testers conducted a thorough evaluation by assessing the model's predictions against a diverse set of data, including both standard and complex edge cases. The success in manual testing provided an additional layer of assurance, emphasizing the model's reliability and superior performance compared to previous models.

Looking ahead, future work may include expanding the dataset to cover a broader range of UML diagrams, including additional types and variations that were not part of the current study. Another important area for improvement is handling incorrect or incomplete diagrams, which could further enhance the robustness of the model. Additionally, integrating the model into an educational tool could provide valuable support for professors and students in software engineering courses, making it easier to teach and learn about UML diagram classification and analysis.

Author Contributions: Conceptualization, I.-G.N. and A.D.I.; methodology, I.-G.N.; software, I.-G.N.; validation, I.-G.N.; data curation, I.-G.N.; writing—original draft preparation, I.-G.N.; writing—review and editing, I.-G.N. and A.D.I.; visualization, I.-G.N.; supervision, A.D.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study were derived from the following resources available in the public domain: <https://github.com/LibreSoftTeam/2016-uml-miner>, <https://www.uml-diagrams.org/>, <https://www.omg.org/spec/UML/2.5/PDF> (accessed on 10 October 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Gosala, B.; Chowdhuri, S.R.; Singh, J.; Gupta, M.; Mishra, A. Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network. *Appl. Sci.* **2021**, *11*, 4267. [CrossRef]
2. Tavares, J.F.; Costa, Y.M.G.; Colanzi, T.E. Classification of UML Diagrams to Support Software Engineering Education. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Melbourne, Australia, 15–19 November 2021. EasyChair Preprint no. 6916.

3. Rigou, Y.; Khriess, I. A deep learning approach to UML class diagrams discovery from textual specifications of software systems. In *Intelligent Systems and Applications*; (IntelliSys 2022); Springer International Publishing: Cham, Switzerland, 2022.
4. Hwang, G.; Jeong, J.; Lee, S.J. SFA-Net: Semantic Feature Adjustment Network for Remote Sensing Image Segmentation. *Remote Sens.* **2024**, *16*, 3278. [CrossRef]
5. Xie, H.; Zhang, L.; Lim, C.P.; Yu, Y.; Liu, H. Feature Selection Using Enhanced Particle Swarm Optimisation for Classification Models. *Sensors* **2021**, *21*, 1816. [CrossRef] [PubMed]
6. Zhang, N.; Donahue, J.; Girshick, R.; Darrell, T. Part-based R-CNNs for Fine-grained Category Detection. In Proceedings of the European Conference on Computer Vision (ECCV), Zurich, Switzerland, 6–12 September 2014; pp. 834–849. Available online: <https://arxiv.org/abs/1407.3867> (accessed on 28 May 2024).
7. Nedelcu, I.-G.; Opranescu, V.; Chiriac, B.-N.; Ionita, A.D. Educational support for automated classification of UML diagrams using machine learning. In *Generative Intelligence and Intelligent Tutoring Systems*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 231–245. [CrossRef]
8. Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Housby, N. An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale. In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, 4 May 2021; Available online: <https://openreview.net/pdf?id=YicbFdNTTy> (accessed on 28 May 2024).
9. Wu, F.; He, J.; Cai, L.; Du, M.; Huang, M. Accurate multi-objective prediction of CO₂ emission performance indexes and industrial structure optimization using multihead attention-based convolutional neural network. *J. Environ. Manag.* **2023**, *337*, 117759. [CrossRef] [PubMed]
10. Object Management Group. *Unified Modeling Language (UML)*; Version 2.5.1; OMG Document Number: Formal/2017-12-05; Object Management Group: Needham, MA, USA, 2017.
11. UML Diagrams. Available online: <https://www.uml-diagrams.org/> (accessed on 10 May 2024).
12. Robles, G.; Fernández, M.A.; Pérez, J.; González-Barahona, J.M. An extensive dataset of UML models in GitHub. In Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, 20–21 May 2017; pp. 519–522. [CrossRef]
13. Robles, G. GitHub UML Study Scripts. *LibreSoftTeam/2016-uml-miner*. GitHub. Available under GPLv2 or Later License. 2016. Available online: <https://github.com/LibreSoftTeam/2016-uml-miner> (accessed on 10 May 2024).
14. Google Images Search. (n.d.). *Google Images Search: A Python Library for Searching Images Using the Google Custom Search API*. PyPI. Available online: <https://pypi.org/project/Google-Images-Search/> (accessed on 10 May 2024).
15. OpenCV. About. OpenCV. 2024. Available online: <https://opencv.org/about/> (accessed on 1 August 2024).
16. Vijay, K.; Priyanka, G. Importance of statistical measures in digital image processing. Mean, Standard Deviation. *Int. J. Emerg. Technol. Adv. Eng.* **2012**, *2*, 56–62.
17. Raschka, S. *Python Machine Learning*; Packt Publishing Ltd.: Birmingham, UK, 2015.
18. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
19. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
20. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the International Conference on Machine Learning, Lille, France, 6–11 July 2015; pp. 448–456.
21. Choi, E.; Bahadori, M.T.; Schuetz, A.; Stewart, W.F.; Sun, J. Doctor AI: Predicting clinical events via recurrent neural networks. In Proceedings of the Machine Learning for Healthcare Conference, Los Angeles, CA, USA, 19–20 August 2016; pp. 301–318.
22. Ding, X.; Zhang, Y.; Liu, T.; Duan, J. Deep learning for event-driven stock prediction. In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, Buenos Aires, Argentina, 25–31 July 2015.
23. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
24. Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Netw.* **2015**, *61*, 85–117. [CrossRef] [PubMed]
25. Chollet, F. Keras. 2015. Available online: <https://keras.io> (accessed on 10 May 2024).
26. Prechelt, L. Early stopping—But when? In *Neural Networks: Tricks of the Trade*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 55–69.
27. Masters, D.; Luschi, C. Revisiting small batch training for deep neural networks. *arXiv* **2018**, arXiv:1804.07612.
28. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
29. Ruder, S. An overview of gradient descent optimization algorithms. *arXiv* **2017**, arXiv:1609.04747.
30. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016.
31. Nedelcu, I.-G.; Caramihai, S.I.; Mocanu, S.; Ionita, A.D. UML diagrams classification with MIMO and SISO neural networks. *UPB Sci. Bull. Ser. C Electr. Eng.* **2023**, *85*, 89.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.