Reverzno inženjerstvo dijagrama pomoću računalnog vida

Šikač, Patrik Noah

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:451101

UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Patrik Noah Šikač

DIAGRAM REVERSE ENGINEERING USING COMPUTER VISION

MASTER'S THESIS

Varaždin, 2022

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Patrik Noah Šikač

Student ID: 0016130318

Programme: Information and Software Engineering

DIAGRAM REVERSE ENGINEERING USING COMPUTER VISION

MASTER'S THESIS

Mentor:

Bogdan Okreša Đurić, PhD

Varaždin, September 2022

Patrik Noah Šikač

Statement of Authenticity

Hereby I state that this document, my Master's Thesis, is authentic, authored by me, and that, for the purposes of writing it, I have not used any sources other than those stated in this thesis. Ethically adequate and acceptable methods and techniques were used while preparing and writing this thesis.

The author acknowledges the above by accepting the statement in FOI Radovi online system.

Abstract

This thesis describes the use of computer vision to recognize shapes and text from images depicting diagrams. The theoretical part of the thesis introduces basic concepts that serve as a foundation to the implementation of the practical part of the thesis. The practical part of the thesis consists of developing a Web application, which comprises a Web client and server. The client is a diagram editor capable of sending an image to the server to be interpreted. Images sent to the server are analyzed with OpenCV, a computer vision library, and a convolutional neural network model.

Keywords: computer vision, machine learning, artificial intelligence, REST API, convolutional neural networks, OpenCV

Table of Contents

1. Introduction

Artificial intelligence is a domain that has been steadily improving over the years and proving that technology is rarely stagnating. It has become so commonplace that it is even being used in consumer electronics, household appliances, traffic, etc. The original idea behind artificial intelligence is simulation of human behaviour with the help of computers. The human ability to see and recognize their surroundings is such an amazing yet undervalued ability. It becomes apparent just how much it is taken for granted when it is tried to be replicated by AI. The subfield that attempts to do that is called computer vision.

There are many notable uses for computer vision today, such as autonomous vehicle driving, face recognition and disease detection. The aim of this thesis is creating a prototype Web application capable of analyzing a diagram and discern various elements it has, then recreating it in a diagram editor. The idea for this thesis actually came from a series of inconveniences that happened while using a graphing platform called draw.io. On a few occasions, caused particularly by forgetfulness, the blueprints for the diagrams were not saved and some corrections to the diagrams had to be made. Thus, the diagrams had to be recreated from scratch and the idea to recreate the diagrams from image files was born from that.

The first part of the thesis serves as an introduction to the domain by explaining the rudimentary concepts of artificial intelligence, machine learning and computer vision. Neural networks are also explored as well as convolutional neural network which are used in the service on the web server. The architecture of the solution is presented along with concepts needed to make it more understandable. Finally, by introducing an use case, the implementation of both the Web client and API are described.

2. Methodology and Work Techniques

As the aim of this thesis was creating a fully-working prototype of an application that uses computer vision to recreate an diagram from an image, it can be concluded that this was an *experimental research*. The environment in which the prototype was developed consisted of a laptop with the operating system Ubuntu installed, as it was concluded that it has better features necessary for software development. Moreover, the development of the prototype and work on research lasted from April to September with brief pauses in June, July and August. The procedure to actualize the research started with a plan of the software architecture. Afterwards, it was researched what technologies provide the utilities essential to develop the prototype. Before conducting the experiment, research into the domain was made to get familiar with the terms of artificial intelligence, machine learning and computer vision. The development on the application started after the research into the domain and upon the end of development, evaluation of the developed prototype was made to determine if the result is satisfactory.

The technologies used to develop the prototype of this thesis:

- Visual Studio Code - an open source source-code editor with support for many programming languages.
- draw.io - an open source graph drawing platform

2.1. Scope and Limitations

Since the goal of the thesis is the development of a prototype for reverse engineering of diagrams, the scope and limitations of the thesis need to be defined. The prototype is developed as a web application that consists of a web client and server that communicate with the help of REST. The web client consists of a diagram editor capable of making rudimentary shapes such as circles, triangles and rectangles. Furthermore, the client is capable of loading an image which is sent to the server. The client's main purpose is to send images and recreate them from the returned data, so the features are kept basic. On the other hand, the Application Programming Interface (API) to which the image is sent to is capable of only running one command at a time, this limitation being so the application was kept simple. The dataset used to create the convolusional neural network model is capable of only detecting triangles, rectangles, circles and stars. However, the diagram interpreter inside the API is expandable and the only requirement is a sufficiently big dataset of a certain shape. Further limitations to the input image the API recieves are as follows:

- Connections between shapes must be non-directional and only one connection is allowed between two shapes
- If the shapes are of the color white, the borders of the shape must be thicker.
- Shapes cannot contain other shapes inside them.

- The labels or descriptions that are inside shapes must have smaller font size, while those outside the shapes should lie outside the connections between the shapes.

3. What is Computer Vision?

Almost every living being capable of experiencing visual sense uses it to perceive the world and its surroundings in a certain manner. For example, bees determine which flowers they shall fly to next, birds look for pieces of bread and seeds on the ground, and humans buy fruits and choose the best ones just by looking at them. Recognizing if a fruit is ripe, unripe or rotten is a seemingly simple action done so quickly that it is almost automatic, yet when attempting to emulate that action using a computer, it is made apparent that even object recognition is a formidable task for machines.

How does the human brain recognize objects? Firstly, eyes [1] pick up visual data that is transported from the retina through the optical nerve, and then into the brain where the data is transformed into information about objects and scenes in the surroundings. MIT researchers have found evidence that object detection is being performed in the inferotemporal cortex. The question is how can that process be accomplished by a machine? The answer lies in computer vision, a specialized field of artificial intelligence. As it can be seen in figure 1 [1] , computer vision [2, p. 1] has the goal of imitating human vision capabilities by mimicking the human sensory and cognitive system, therefore providing methods for image formation and machine perception of the given image.

While the implementation of the human sensory system is focused on hardware and the design and proper placement of cameras, the imitation of the human cognitive system is primarily realized using machine learning. Nevertheless, it is important to mention that there are many ways to implement visual perception of a machine. Before the advancements in the field of machine learning, the methods [2, p. 2] used for extraction of information from photographs consisted of denoising, edge finding, texture detection and shape-based operations. Those methods were implemented using expert systems, a subfield of AI that has the intention of reproducing the way humans think, in other words, human logic.

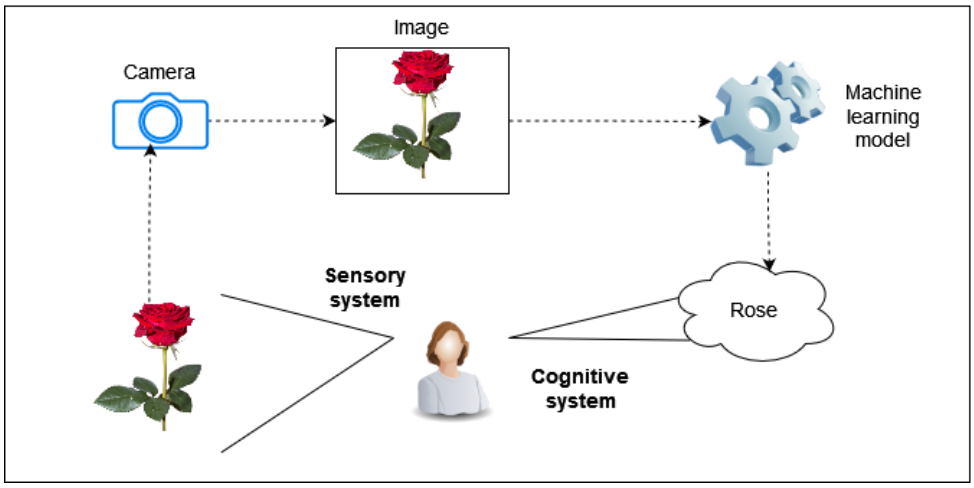


Figure 1: Comparison of human visual perception and object recognition with that of a machine; according to [2, p. 2]

[1]The photo of the rose was acquired from: https://purepng.com/public/uploads/large/purepng.com-roseflowers-rose-red-rose-961524681182nldfx.png

3.1. Artificial Intelligence

Since machine learning is a crucial part of computer vision, it is deemed necessary to explain what artificial intelligence is, as these two terms are closely related. Britannica encyclopedia [3] defines artificial intelligence as the ability of a computer or a computer-controlled robot to perform tasks commonly done by humans because they require human intelligence or discernment. As of now, no AI can match the ability to perform the wide variety of tasks humans can do, but they can match or even exceed humans performance in some specific assignments. There are many areas of intelligence that are replicated using AI; some of them are [3]:

- Learning - The ability to learn through past experiences. It is usually combined with other areas of intelligence.
- Reasoning - Programming a computer to form a conclusion based on the given data has been successful, but the perfection of it still remains a challenge.
- Perception - The capacity to understand the given stimuli and to successfully interact with it.
- Problem solving - It can be defined as a systematic search through a range of possible actions in order to reach some goal.
- Language - The capability to recognize a human language and to respond competently to the given questions and answers.

3.2. Machine Learning

In the previous chapter, it is mentioned that the capability to learn is an area of artificial intelligence. Machine learning [4, p. 2] is a multidisciplinary field, as it draws from on results from artificial intelligence, probability and statistics, computational complexity theory, control theory, information theory, philosophy, psychology, neurobiology, and other related fields . Lakshmanan [2] defines machine learning as a sub-field of AI that teaches AI how to react to certain inputs by feeding it a large amount of data and instructing them to learn from it. Thus, it can be said that the key feature of machine learning is the ability to learn from previous experience. Mitchell [4, p. 4] states that, for a computer program to be able to learn from experience, while performing a series of tasks, the measured performance must improve with experience. For every learning problem it is necessary to identify the following features [4, p. 3]:

- Task
- Performance measure
- Training experience

Following that definition, within the domain of this project, the task of the learning problem is the recognition and classification of detected geometrical shapes inside an image. The

performance measure is the ratio of correctly classified shapes and the training experience is a large set of N various geometrical shapes $\{x_1, \ldots, x_N\}$ that is called a *training set* [5, p. 2]. The training set is used to configure and build a model that will be used to classify the shape from the image. The images inside the training set are usually classified by checking them individually and hand-labeling them accordingly. For it to be understandable to machines, the classification of an image is represented by a *target vector* $\boldsymbol{t}$.

Furthermore, the common practice is to preprocess the input value as it increases the accuracy of the model, thus making it easier to correctly categorize the shape in the image. Preprocessing is a crucial step in machine learning process that takes raw input data and transforms it into a uniform and clean format understandable to machines [6]. It consists of many disciplines, as data preparation and data reduction techniques [7]. Data preparation includes data transformation, integration cleaning and normalization, while data reduction has the goal of data complexity reduction with the use of feature selection, instance selection or by discretization. Preprocessing for the project is explained in chapter 6.

There are three approaches to machine learning: supervised, unsupervised and reinforcement learning [5, p. 3]. In supervised learning, the input values in the training set have defined corresponding target vectors. Unsupervised learning is the exact opposite of that; the training set doesn't contain any target vectors. This type of learning is used mostly for knowledge discovery and data visualization. Lastly, reinforcement learning has the aim of discovering the optimal set of actions to fulfill a goal. One of its uses is training a video game AI to play the game optimally.

Many algorithms can be used to build a machine learning model. Some of them are decision trees, hidden Markov models, Bayesian networks, Support Vector Machine, K-Means Clustering, Neural Networks, etc. For the purpose of this project, only one algorithm is interesting: the neural network algorithm. For some specific types of problems, such as learning to interpret real-world sensor data, neural networks have a reputation as one of the most effective learning algorithms currently known [4, p. 81]. In the past, they have successfully learned how to recognize handwritten characters, spoken words and faces.

In the domain of computer vision, the classification model is often implemented using convolutional neural networks (CNN) and it will be so in this project. Convolutional neural networks [8, p. 240] are a type of neural networks that are most suited for image classification and use deep learning to create a model. But, before it is explained how convolutional neural networks work, it is necessary to clarify what neural networks are.

3.3. Neural Networks

Haykin [9, p. 24] characterizes a neural network as a machine that is designed to model the way in which the brain performs a particular task or function of interest. At the core of the neural network lies the artificial neuron, a computing cell that is used to achieve an interconnected adaptive learning machine. There are many artificial neuron models, but the first and the most basic one is called a *perceptron* [10] and it was developed through the 50s

and the 60s of the 20th century by Frank Rosenblatt, a scientist who was inspired by the earlier work of Warren McCulooch and Walter Pitts . Through the years, the perceptron neuron was replaced by other types of neurons, the most common one being the *sigmoid* [10] neuron.

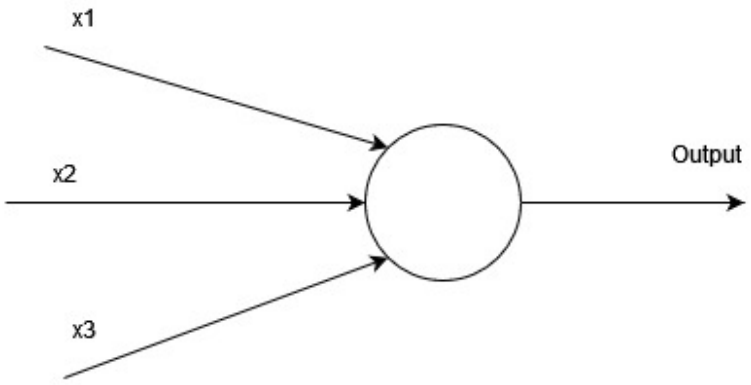


Figure 2: Perceptron structure; according to [10]

In figure 2, it can be seen that a perceptron takes several inputs x_1, x_2, x_3 and produces a single output. The output, which can be either 0 or 1, is determined by the *weights* of the inputs w_1, w_2, w_3. Each weight is a number which expresses the importance of the respective inputs to the output. Each of the weights is multiplied with the value of their corresponding input and the products are summed up. The sum of weights $\sum_j w_j x_j$, that is also written as a dot product $w \cdot x$, is compared to some threshold value. The threshold is also called the *bias* and is a real number that determines how easy it is to output the value 1. The greater the bias, the easier is it for the perceptron to output a 1. The output is determined by the following expression [10]:

$$f(x) = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{3.1}$$

The perceptor can be better explained with an example. A customer wants to buy a bluetooth speaker and so he goes to the store where he finds a speaker that catches his eye. The customer has three factors that will determine whether he will buy it or not:

- 1. Is the price under 100 euros?
- 2. Does the battery last more than five hours?
- 3. Is the sound quality adequate?

In this scenario, the speaker is above the price range, but the battery lasts more than five hours and the sound quality is satisfactory. The factors can be represented as binary variables with the following values set $x_1 = 0, x_2 = 1$ and $x_3 = 1$. However, the three factors do not have

the same influence, or in this case, weight, on the decision whether to buy this speaker or not. This customer is a little bit tight with money so the weight for the first factor is $w_1 = 5$. Secondly, he thinks sound quality is more important than battery life, so the weights for those factors are $w_2 = 2$ and $w_3 = 3$. Lastly, the bias for this perceptron equals $b = 6$. The output is calculated as follows:

$$\begin{aligned} f(x) &= w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b \\ &= 5 \cdot 0 + 2 \cdot 1 + 3 \cdot 1 + 6 \\ &= 0 + 2 + 3 + 6 = 11 > 0 \\ &= 1 \end{aligned} \tag{3.2}$$

From the output, it can be concluded that the customer will buy the speaker. This kind of output calculation was an example of a certain type of activation function [9] called threshold function. There are two other activation functions that are relevant to this project: the rectified linear unit (ReLU) and softmax. In short, ReLU is an activation function mostly used for deep learning that works similar to the treshold function, with the difference being it returns the largest input value:

$$f(x) = x^+ = max(0, x) \tag{3.3}$$

The softmax function [11, p. 180, 181] is used to represent a probability distribution over a discrete variable with n possible values. They are most often used as the output of a classification model, to represent the probability distribution of n different classes. It is a generalization of the sigmoid function, an activation function that is utilized to represent a probability distribution over a binary variable. The softmax function is expressed with:

$$softmax(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{3.4}$$

where $z_i = log\check{P}(y = 1|x)$. This is because the number needs to be between 0 and 1, and the logarithm of the number needs to perform well in the gradient-based optimization of the log-likelihood. Furthermore, the exponentiation and normalization results in a Bernoulli distribution controlled by the sigmoid function.

3.3.1. Network Layers

A single neuron is not enough to create a neural network. In fact, an artificial neural network (ANN) [8, p. 109] consists of many neurons that are connected to each other. Each output of a neuron is connected to many other neurons in the network. To achieve good performance and solve complex problems, most neural networks are created using MultiLayer Pereptrons (MLP). As seen in figure 3, neural networks have neurons organized into three or more layers, where each connection between the layers goes into one direction, forward.

This type of neural networks are called *feedforward* neural networks. The first layer is

the input layer; it consists of n number of neurons that correspond to the input parameters, e.g. number of pixels in an image. On the other end of the neural network lies the output layer, where the number of neurons is equal to the number of expected outputs, or classes. Finally, the last layer is the hidden layer. The main use of the hidden layer is to intervene between the input layer and the hidden layer in some useful manner [9, p. 43]. The hidden layer can actually consist of more than one layer, and by doing that, the network gains the opportunity to extract higher-order statistics. This ability is especially useful when the size of the input layer is rather large.

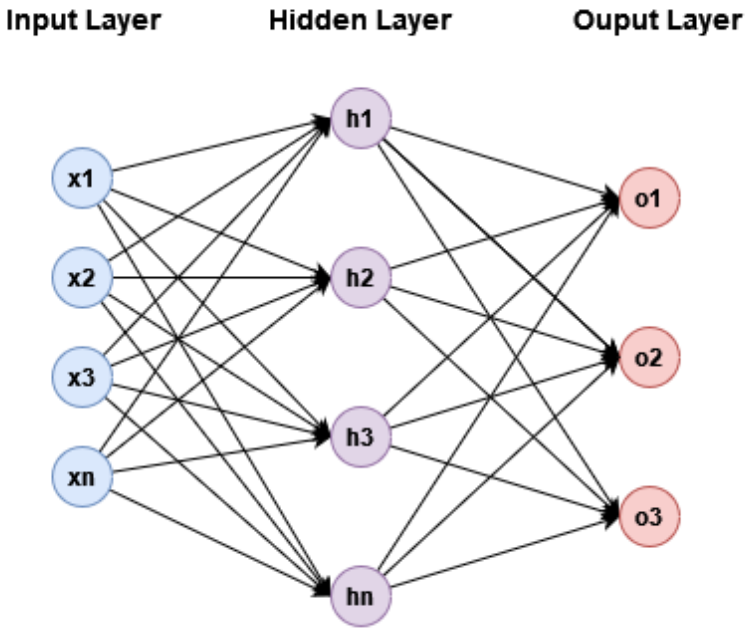


Figure 3: MultiLayer Perceptron Neural Network; according to [5]

3.3.2. Learning Algorithm

The previous sections covered the architecture of neural networks, but the main and the most important question remains unanswered: How do they learn? Neural networks have two operating modes [8, p. 113]: the already mentioned feedforward or testing mode, and supervised learning. As mentioned before, feedforward consists of inserting a pattern to the input layer that is processed into information by neurons and then propagated through the network where the output neurons produce a result. Supervised learning is a process in which the input layer is presented a pattern and the weights of neurons need are adapted so they return a value as close to the target value as possible. For supervised learning, one of the simplest and most general methods is called the *backpropagation* algorithm [8, p. 113].

The algorithm starts with the initial execution of feedforward where the initial outputs of all neurons in each of the layers is calculated. The outputs [10] of the neurons $y(x)$ for all training inputs x are compared to true values a. With the help of an error function, the deviation is evaluated for each output neuron and for each training set sample. This function is also

called a loss function [10], cost function or an objective function. The result of the evaluation of the overall deviation is a scalar value dependant on the initial values of all the network weights w and biases b which must be updated adequately during the learning process in order to minimize the error function. To make that possible, the quadratic cost function, also known as the mean squared error (MSE) [10], is used to evaluate all the output neurons related to each training set sample, all while minimizing the subsequent cost function $C(w, b)$ in respect to the weights and biases of each neuron:

$$C(w, b) = \frac{1}{2n} \sum_{x} ||y(x) - a||^2 \tag{3.5}$$

In the equation, $||v||$ stands for the length function for a vector v. Furthermore, while looking at the cost function, it can be concluded that $C(w, b)$ is non-negative, because every argument in the sum is non-negative. Secondly, the cost $C(w, b)$ gets to the value $C(w, b) \approx 0$, just as the value of $y(x)$ gets approximately equal to the expected output a, for all training inputs x. That means the weights and biases have been correctly configured and the neural network is learning successfully. On the other hand, a large $C(w, b)$ means that $y(x)$ is not close to the expected output a for a significant number of inputs. It can be concluded that the aim of the backpropagation algorithm is to get the loss function as small as possible by finding a certain set of weights and biases for all neurons.

3.4. Convolutional Neural Networks

Although neural networks are great in a lot of use cases, they are not particularly great for computer vision; the main reason being that an input for computer vision are the pixels from an image, and images can hold a great amount of pixels, which in turn can cause a significant toll on the performance of the neural network. The solution for this are convolutional neural networks [9, p. 267] (CNN), a special class of multilayer perceptrons. Convolutional neural networks are also an example of deep neural networks. These kind of networks are characterized by having more than one hidden layer [2, p. 43], therefore increasing the number of trainable parameters.

When using neural networks for image classification, where the input is a 28×28 image [12], each pixel's intensity is used as a value of a input neuron. That means that the neural network will have $28 \times 28 = 784$ input neurons. During classification, it is important for the class to be invariant under translations, scaling, rotations, as well as elastic deformations [5, p. 267]. Furthermore, in a fully connected neural network architecture, spatial structure of the images is not taken into account. Thus, pixels that are close together and far apart are treated the same. Convolutional neural networks use the spatial structure to their advantage by extracting *local* features that make up only a small part of the image. Features [12] are a certain configuration of inputs that will cause the hidden neuron to activate. The information about local features can be merged in subsequent layers of the network in order to detect higher-order features, which ultimately results in information about the image as a whole object. Moreover, the local features might prove to be useful in other regions of the image in a situation where the object of

interest was translated. The whole concept of convolutional networks is based on three ideas [12]: *local receptive fields*, *shared weights*, and *pooling*.

3.4.1. Local Receptive Fields

When working with images, one should take into account the dimensions of the images [8, p. 242]. Images actually have a volume $W_0 \times H_0 \times D_0$, because the depth of the image corresponds to the number of channels of the image, i.e. RGB where $D_0 = 3$. Continuing with the 28×28 image example with the assumption that the image has only one channel, the input neurons can be imagined as a square of neurons, as shown in figure 4. Normally, every neuron from the input layer would be connected to every neuron in the hidden layer. That is not the case with the convolusional networks. Instead, a small region of the input image is extracted and connected with a single neuron of the hidden layer. That way, a local feature is extracted and its position becomes less important, as long as its position to other features is preserved [9, p. 267]. The neuron learns the overall bias of the field along with the weight, and once the neuron analyzes the receptive field, the local receptive field moves a certain amount across the image. That movement is called *stride* [13] and it follows the top to bottom, left to right principle. It should be noted that stride is usually one or two pixels, but it can be more in certain situations.

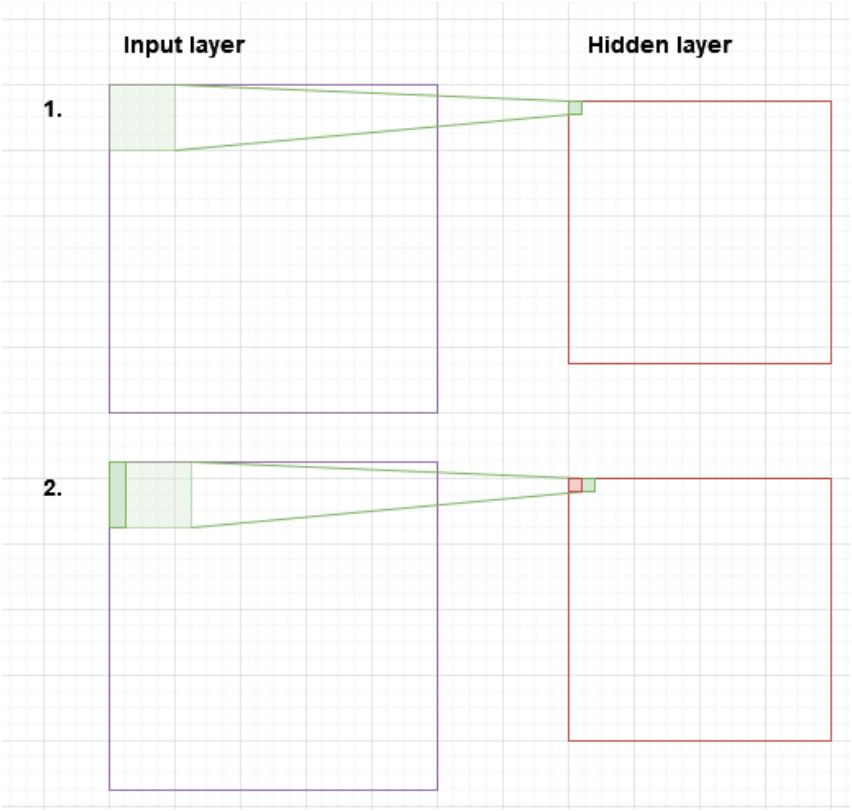


Figure 4: Feature extraction using local receptive fields with a stride length of one pixel; according to [12]

3.4.2. Shared Weights and Biases

With receptive fields with size 5×5 and the stride length of one pixel, the result of feature extraction is a hidden layer the size of 24×24 neurons. Each of the hidden neurons in the layer share the same weight and bias, which means all neurons in the layer detect the same feature [12]. For instance, if the weight and bias of a neuron are configured to detect a corner or an edge in a local receptive field, making the bias and the weight shared across the layer would result in the ability to detect those features anywhere in the image. Thus, the result of mapping the features from the input layer to the hidden layer a *feature map*. With the use of weight and bias sharing, the number of free parameters [9, p. 267] is reduced, making the performance of the network better. It should be also emphasized that all weights and biases in all the layers of the convolutional network are learned through training automatically; in conclusion, the network learns on its own which features to extract.

The shared weights and the shared bias of the feature map define a *kernel* [12] or a *filter*. Since a feature map can only detect one feature, to have a successful object recognition, many more feature maps are needed. A group of feature maps constitute a *convolutional layer*. The number K of convolutional filters [8, p. 243] is normally chosen as a power of number 2, such as 32, 64, and 128.

There is an additional parameter that can be set for the convolution process - *padding*. It is a operation that adds a border around the input image. The border is made up from zeroes and it is used to preserve the dimension of the input image when creating feature maps. The dimensions of the output feature maps is calculated as follows [8, p. 243]:

$$W_1 = \frac{(W_0 - w + 2P)}{S} + 1 \qquad H_1 = \frac{(H_0 - w + 2P)}{S} + 1 \tag{3.6}$$

The equation argument P is the padding parameter, S represents the stride length, and w is the dimension of the input image if the image has dimensions $w \times w \times D_0$. Consecutive to the mapping operation, the dimensions of the feature maps are $W_1 \times H_1 \times K$, where K is equal to the number of convolutional filters.

3.4.3. Pooling

Each convolutional layer is usually followed by a computational layer that reduces the resolution of the feature maps. The reduced resolution [9, p. 267] has the effect of reducing the sensitivity of the feature maps' outputs to shifts and other forms of distortion. This operation is called *pooling* where the output of the convolutional layer is condensed by checking a local receptive field and taking one activation from the feature map. The size of the receptive field is not too big, commonly being 2×2 or in some cases 3×3. There are two approaches [13] to this operation: *max pooling* and *average pooling*.

The max pooling approach consist of taking the maximum value in the receptive field and mapping it to a new map, moving by the value of the stride length, and repeating the step until all the values have been mapped. On the other hand, average sampling functions by

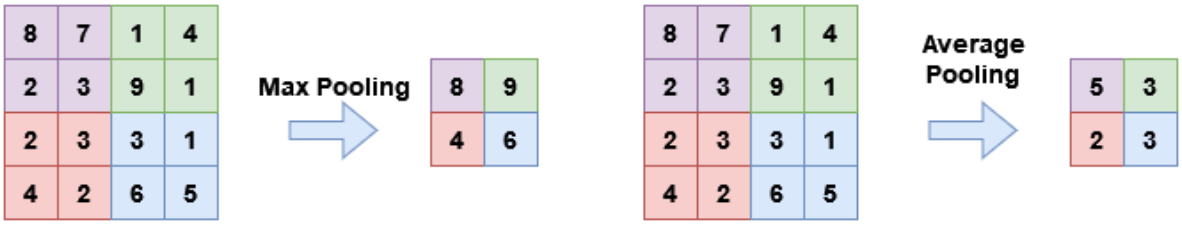


Figure 5: Max and Average Pooling Example with stride length of two; according to [8]

calculating the average value of the receptive field and mapping it. An example of how pooling works can be observed in figure 5.

3.4.4. Convolutional Neural Network Architecture

Since convolutional neural networks are deep neural networks, they have a complex architecture. They contain multiple kinds of layers, each with their own purpose and order in the architecture as seen in figure 6. The layers used in the architecture are as follows [8, p. 247]: *input layer*, *convolutional layer*, *activation layer*, *pooling layer*, and *fully-connected layer*.

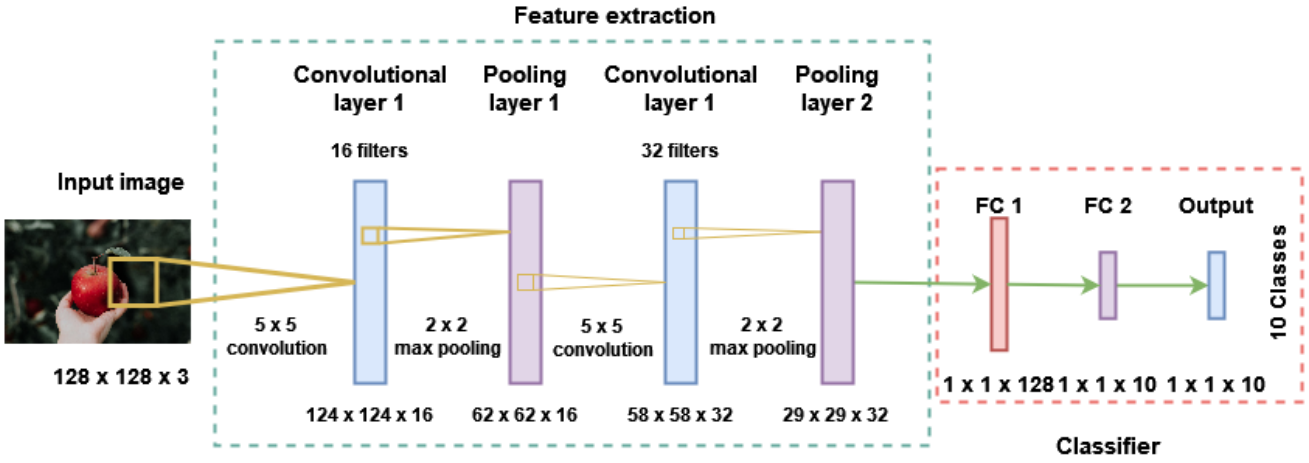


Figure 6: Convolutional neural network architecture; according to [8, p. 247]

Input layer - The input layer has an input image of certain dimensions that is propagated to the convolutional layer.

Convolutional layer - Convolutional layers are used for detecting features from the input image. A convolutional layer outputs several feature maps that contain the information about the detected features. Usually, multiple convolutional layers are used in a neural network. Each following convolutional layer detects more features than the one before. For example, the initial layers detect rudimentary features like edges and corners, while later layers may detect complete features like eyes, ears or noses.

Activation layer - The activation layer always follows the convolutional layer, but it is not

characterized by any parameters and the convolution volume is preserved [8, p. 243]. It applies a certain activation function to the feature map. While there are many activation functions, the most popular one is ReLU.

Pooling layer - This layer is used to subsample each feature, or in other words, reduces the size of feature maps, thus reducing the feature maps' sensitivity to distortion and increasing performance.

Fully-connected layer - By repeating the convolution, activation and pooling layers, the dimension of width W and height H diminish in size, while the depth D of the volume increases. The output of the last convolusion layer becomes the input of the fully-connected layer which is realized as a conventional neural network where each neuron of the previous layer is connected to every neuron in the following layer. Fully-connected layers have a dimension $1 \times 1 \times K$ in where K can represent the number of classes if the FC layer is the last one. That means that K neurons are completely connected to the volume map or previous FC layer.

While the architecture of the convolutional neural network may have been explained, there are two operations that are important in creating an accurate and well performing model and they are *drouput* and *batch normalization* [8, p. 252].

While training a CNN, one must be wary of *overfitting* the model. When training a model, it is possible to train the model to be too specific so that it does not perform well when presented with new data (i.e. validation data). To achieve a higher level of generalization, a regularization method called dropout method is used. The idea behind the method is that by a defined probability p, random neurons are selected in a stochastic manner and excluded from the neural network during a training phase. Exclusion of random neurons is used to prevent the model from becoming excessively adapted at the expense of generalization. This forces the neurons to learn more robust features that are usable in various subsets of neurons. The dropout method is sometimes used between convolusional layers and often between FC layers.

The values of pixels in images are normalized to be in the range [0,1], which is in line with the dynamic range of most activation functions and optimizers. However, after propagating the input through a hidden layer, the resulting output values no longer lie in the required dynamic range for the subsequent layers [2, p. 49]. Consequently, the neuron's output is zero or inaccurate. The solution to this problem is batch normalization, another regularization technique that works by splitting data into small training batches and subtracting neuron outputs with the average value and dividing by the standard deviation. Just by doing that, the distribution of neurons would be far too centered and normally wide, which would result in invariable behavior. That problem is mended by introducing two learnable parameters to the neuron: *scale* and *center*. Afterwards, the input data to the neuron is normalized with:

$$normalized = \frac{input - center}{scale} \tag{3.7}$$

By using this method, the shape of the input data is preserved, but the data is normalized. The direct result of batch normalization is a shorter amount of epochs (training phases) needed to achieve convergence and generalization .

3.5. Real World Applications of Computer Vision

As technology progressed, computer vision has shown great potential to be utilized in many domains and in increasingly complex ways. Through research and real world applications, it has become apparent just how commonplace computer vision is. Interestingly, the major driver of progress was the visual data procured from smartphones, security systems, traffic feeds, etc. It has found use in healthcare [14], entertainment, business, security, military and general life of people.

One of the most obvious and recent examples of real-world application of computer vision are self-driving vehicles. They use cameras and other sensor to send the data to the computer vision software, which in turn is utilized to identify other vehicles in traffic, lane markers, pedestrians, traffic signs, bicycles, and all the other relevant information needed to safely navigate the road. Another example is Google Translate; it grants users the ability to scan and translate text from photos or in real-time. Google also implemented the reverse image search for their web search engine; it allows users to input an image and the algorithm returns search results best-matched to the given input image. Social media applications [15] like Snapchat, Instagram, Facebook and others use face-detection to recognize people in images and apply filters to their face map. Biometrics like fingerprints and iris recognition are prevalent methods used for authentication to access mobile phones and laptops.

Likewise, there have been many innovations in medicine. Pattern recognition powered by AI, machine and deep learning has numerous uses in pulmonary medicine [16, p. 6] where predictive models were constructed with the ability to detect signs of lung cancer, molecular signatures for diagnosis of pulmonary fibrosis and COVID-19. Computer vision has also been used in other subfields like cardiology [17, p. 1], pathology, dermatology, ophthalmology.

Additionally, there have been interesting application examples of computer vision in the military domain. Milisavljevic [18] has researched various methods of mine detection utilizing shape recognition from visual data. The visual data was obtained from infrared images of the mines and then preprocessed into binary images. The conclusion of the research was that all methods tested were viable depending on the quality of the visual data and the preprocessing. Svenmarck et al. [19, p. 3] mention use of AI for maritime surveillance using fixed radar station, patrol aircrafts, ships and automatic identification systems. Secondly, underwater mine searches are more and more performed by autonomous underwater vehicles (AUV). Those vehicles are equipped with synthetic aperture sonars, which return acoustic imagery of the sea floor, which shows potential for use of deep neural networks. Lastly, a lot of controversion runs around the increasing development of lethal autonomous weapon systems (LAWS) which also use visual data to operate and carry out their assignments. They have also been given a nickname "killer robots" because of their ethical implications.

4. The Architecture of the Solution

In this chapter the architecture of the solution is explained so it is clear how the prototype works. With the help of a diagram seen in figure 7, the relation between each of the modules is clarified. Moreover, the chapter features explanations of the roles each individual part has in the architecture. The implementation of the solution this architecture presents is available on GitHub[1]. But first, there are some concepts that need to be clarified before breaking down the architecture of the solution: API, HTTP and RESTful architecture.

API [20] stands for **A**pplication **P**rogramming **I**nterface, a set of rules that explains the communication rules and protocols between applications or computers. API is what stands between a web based client and the server. This interaction consists of a few steps, the first being the initiation of an API call from the client. Those requests are made to the web server by sending the request to a specific Universal Resource Identifier (URI), alongside a request verb, header and in certain cases, body. If the request is valid, the API propagates the request to the server or an external program. After the request has been processed, the server sends a response to the client with required information. Finally the data is transferred to the caller by the API.

HTTP [21] stands for **H**yper**T**ext **T**ransfer **P**rotocol, the foundation of every data exchange on the Web. It is a client-server protocol that is used to fetch resources like HTML documents. The communication is always started by the client, generally a Web browser, by sending a request to the server. The result is a Web page that consists of many sub-documents such as text, images, videos, scripts, etc. Requests and responses are exchanged as individual messages that contain packets of information. Requests are sent by Web browsers, while the responses are given by Web servers. It should be noted that the communication between the client and the server is generally not direct. In fact, there are normally proxies that handle the requests and responses before they arrive to their final destinations. Some real-world examples of proxies are modems and routers. HTTP requests are discerned by their methods (GET, POST, DELETE, PUT) that define what kind of action the client wants to carry out. Aside from methods, requests also contain headers, the path to the resource and the body which contains some information. On the other hand, responses are discerned by their status codes and messages. Responses also contain headers and bodies.

As mentioned before, there must be a diagram editor to display and modify diagrams, and an module that is capable of discerning objects from an image. The solution is divided into two main components: the REST client, and the REST API. REST [22] is an acronym that stands for **RE**presentational **S**tate **T**ransfer and it is an architectural approach for a *distributed hypermedia solution*. It was presented by Roy Fielding [23] in his dissertation in the year of 2000. This architectural style is defined by six principles and an interface that satisfies those principles is named *RESTful*.

The six principles are as follows [22]: *uniform interface, client-server, stateless, cacheable, layered system, code on demand*. A uniform interface ensures that the interface for resources

[1] URL to the thesis implementation: `https://github.com/psikac/diagram_reverse_engineering`

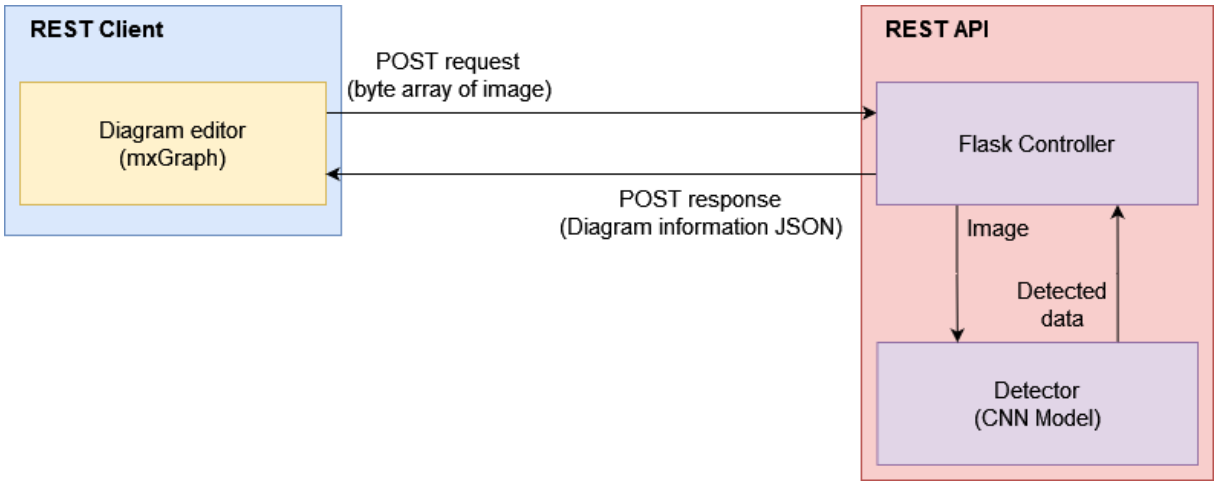


Figure 7: The architecture of the solution [Author's work]

provided by the API must be defined and in a way that the URI for a resource is unique. The resource also must not be too large and should be uniformly formatted (naming convention, URI naming convention, data format). The client-server design is used to prevent the client and server from becoming intertwined, thus allowing them to evolve separately. By introducing statelessness, the server is forbidden from saving any context information about the client or its requests, meaning that the requests sent from the client have to contain all the information needed to process it. The responses from the server can be later cached by the client and reused, but the responses have to state whether they are cacheable or not; this is mandated by the cacheabilty principle. A layered system is one that has a hierarchical structure that prevents layers from accessing non-adjacent layers. Finally, code on demand principle allows extension of the client functionality with scripts and applets. Servers provide them and clients only have to download the scripts or applets and then execute them.

The REST client consists of a web page that contains the diagram editor which is implemented with a library called mxGraph. The diagram editor is used to draw diagrams, edit them and import them from images. By using the option to import a diagram from an image, the selected image is parsed as a byte array that is then sent to a REST API's endpoint. Afterwards, the data is sent using a POST request to the endpoint. The endpoint is a controller provided by Flask, a Python framework that is used to create backend for web applications. The controller forwards the resource, in this case a byte array of an image, to the diagram interpreter, a machine learning model implemented with a convolutional neural network. Inside the interpreter, the image is analyzed and the model recognizes objects from the image. Subsequently, the data is formatted as JSON and propagated to the Flask controller, that in turn, sends the response to the REST client.

5. Diagram Interpreter API

This chapter contains the explanation of how the diagram interpreter API was developed and how it works. The first section explains how the API communicates with the web client and how it handles requests. The structure of the interpreter, the approach to object recognition and the progress of achieving it is explained in the second section.

5.1. Technology Behind the API

Before it was decided on which programming language would be used to implement the backend, it was necessary to choose which library would be used for computer vision. In initial phases of planning, it was decided that OpenCV would be used for the project. According to the official site, OpenCV stands for Open Source Computer Vision Library and it is an open source computer vision and machine learning software library [24]. It was produced with the goal of providing a common infrastructure for applications that use computer vision and to advance the use of machine perception in the commercial products. There are more than 2500 optimized algorithms that can be used for various purposes common in such applications, like object identification, classification of human actions in video materials, face identification and recognition, extracting 3D models from objects, etc. Some of the companies that utilize it are Google, Toyota, IBM, Intel and Microsoft. It has support for Windows, Linux, Android and Mac OS and is available in C++ (in which is natively written), Python, Java and MATLAB.

Since both OpenCV and mxGraph have Java support, it was obvious that the backend would be written in Java. However, there were complications with setting up OpenCV to work in Java and it was decided that Python would be the programming language of the backend. Python [25] is defined as an interpreted, interactive, object-oriented programming language. Some of Python's programming paradigms are not characteristic for object-oriented programming, i.e. procedural and functional programming. Guido van Rossum created Python in the late 80s out of necessity when he was working in the Amoeba distributed operating system group at CWI. It is known for its performance, specific syntax and extensive library support. Python's syntax is written in a way that recognizes groups of code by using indentation. The creator of Python states that programs written this way offer more clarity and elegance to the structure of the programs. An example of such code can be seen in listing 1.

Listing 1: Example of finding even numbers written in Python

```
1     list = [1,2,3,4,5,6]
2     for i in list:
3        if(i % 2 == 0):
4             print(f"Number {i} is even")
5         else:
6              print(f"Number {i} is even")
```

5.2. Communication With the Web Client

For the client to be able to communicate with the server which runs the object recognition model, necessary infrastructure must be established. This is accomplished with the RESTful architecture and Flask[1], a small library that empowers Python to create web application. Flask is further extended by FlaskRESTful[2], an extension that provides RESTful capabilites.

The application is separated into parts so their purpose is as simple as it can be and the structure of the solution is clear. The main parts of the application are:

- Core of the application
- Controllers
- Diagram interpreter

Listing 2: Core part of the application

```
app = Flask(__name__)
api = Api(app)
CORS(app)

api.add_resource(ImageController, '/image')

if __name__ == '__main__':
    app.run(debug=True)
```

The central or core part of the application is located in a file by the name of app.py. It has a fairly simple purpose as it is only used to set up the RESTful server and run it. In listing 2 it is shown how the application runs. Firstly, the app is run as a Flask application and it is established that the application runs as an API. Since the web client sends data from another web port, Cross-Origin Resource Sharing (CORS) [26] must be enabled to give the server the permission to load resources from another domain, port or server. For the server to be able to process the request, a controller must be added to the API. As it can be seen in listing 3, the controller checks whether the content from the request is in JSON format; in case it is, the image is extracted from the request and forwarded to the DiagramInterpreter class.

Listing 3: Image controller

```
class ImageController(Resource):

def post(self):
    content_type = request.headers.get('Content-Type')
    if (content_type == 'application/json'):
      data = request.json['image']
      interpreter = DiagramInterperter()
      results = interpreter.get_diagram_elements(data)
      if(results != None):
```

[1]Link to Flask's documentation: https://flask.palletsprojects.com/en/2.2.x/
[2]Link to FlaskRESTful: https://flask-restful.readthedocs.io/en/latest/

```
10              return results
11          else:
12              return 'Content-Type not supported
```

The upcoming sections focus on how the implementation of the diagram interpreter. To make them easier to follow, a scenario was created that is used throughout the forthcoming sections. The web client has sent a request that is received by the image controller. The image is forwarded from the controller to the interpreter so it can analyze the image and return the data to the controller. As seen from figure 8, the image from the request consists of three objects, each a different shape with a label inside.

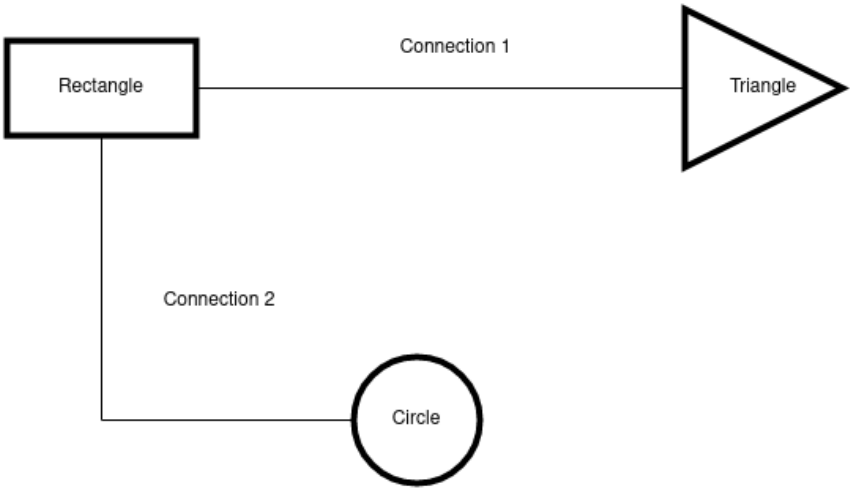


Figure 8: Image sent from the client [Author's Work]

5.3. Implementation of the Interpreter

The interpreter class contains the business logic of the application, in this case object recognition from images. As a component, it is implemented following the Singleton [27] design pattern. In short, implementation of this design pattern ensures that only one instance of this object can exist during run-time of this application. Usually, this would not be used in a real web application, since this way only one request at a time could be processed, and that would cause other users to wait in a queue to process their image. However, for the purposes of the prototype, this design pattern is adequate as requests come from one client.

Listing 4: Diagram interpreter singleton

```
1   class DetectorMeta(type):
2       _instances = {}
3
4       _lock: Lock = Lock()
5
6       def __call__(self, *args, **kwds):
```

```
7               with self._lock:
8                   if self not in self._instances:
9                       instance = super().__call__(*args, *kwds)
10                      self._instances[self] = instance
11              return self._instances[self]
```

From listing 4 the implementation of the Singleton is shown. The implementation was taken from Refactoring Guru[3], a website with information about design patterns. In Python this is accomplished with metaclasses, which are according to the official Python glossary [28], classes of classes which allow programmers a higher level of control of how some classes behave. This specific implementation seen in listing 4 ensures thread safety, so all threads are synchronized at the first access to the Singleton.

The very DiagramInterpreter class has only one method available to outside classes and it is called get_diagram_elements. This class calls other private methods (methods that are only available inside the class) where each method serves as a step in extracting information from the diagram image.

5.3.1. Text Recognition

For the diagrams to be informative, aside from the shapes and connections, they must also have text. Shapes and connections can be detected and recognized using OpenCV computer vision and machine learning classification, respectively. However, OpenCV does not offer optical character recognition (OCR) to detect text from images. For text recognition, Google's OCR engine that is free for use. The implementation of character recognition was based on the tutorial available on GeeksForGeeks[4]

Listing 5: Optical character recognition using Tesseract

```
1       def __detect_text(self, img):
2           custom_config = r'--psm 3'
3           img_grayscale = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
4           results = pytesseract.image_to_data(img_grayscale,config=self.custom_config,
                output_type=Output.DICT)
5           detected_text = []
6           for i in range(0, len(results["text"])):
7               if(len(results["text"][i].strip())):
8                   detected_text.append(Label(results['text'][i], results['left'][i],
                     results['top'][i]))
9           return detected_text
```

In listing 5 it can be seen that the image is first transformed into grayscale for clarity. Afterwards, using the image_to_data method, Tesseract returns all the text it recognizes and relevant data concerning it in a dictionary format. The results are iterated through and all data where the length of the text is longer than 0 is appended to the list of detected text.

[3]The code implementation can be found at: https://refactoring.guru/design-patterns/singleton/python/example

[4]OCR tutorial: https://www.geeksforgeeks.org/text-localization-detection-and-recognition-using-pyte

Furthermore, the data is appended as a Label object which has the following attributes: value, x and y coordinates.

It should be emphasized that the Tesseract API is not always accurate when reading text from an image. The output depends a lot on the quality of the image, the font of the text, placement of the text and its surrounding. For instance, in the example scenario the output of the API that is shown in listing 6 lacks the label placed inside the circle. It also did not correctly recognize the outer label "Connection 1" and parsed it as "Connection 4".

Listing 6: Results of scanning the example image with OCR

```
{'value': 'Connection', 'x': 292, 'y': 61}
{'value': '4', 'x': 357, 'y': 61}
{'value': 'Rectangle', 'x': 67, 'y': 76}
{'value': 'Triangle', 'x': 501, 'y': 86}
{'value': 'Connection', 'x': 142, 'y': 221}
{'value': '2', 'x': 206, 'y': 221}
```

5.3.2. Preprocessing

The first step in the object recognition algorithm is preprocessing of the data. While preprocessing usually involves normalization of data and denoisification, in computer vision the procedure is somewhat different. The input image must be accommodated to the format which OpenCV expects and works best with. It should be pointed out that the approach to image preprocessing and analysis used in this project was found on an OpenCV forum [5]. The solution that the user submitted was programmed in C and thus needed to be adapted to Python. As a result, only the basic idea of the solution was adapted.

Listing 7: Image preprocessing

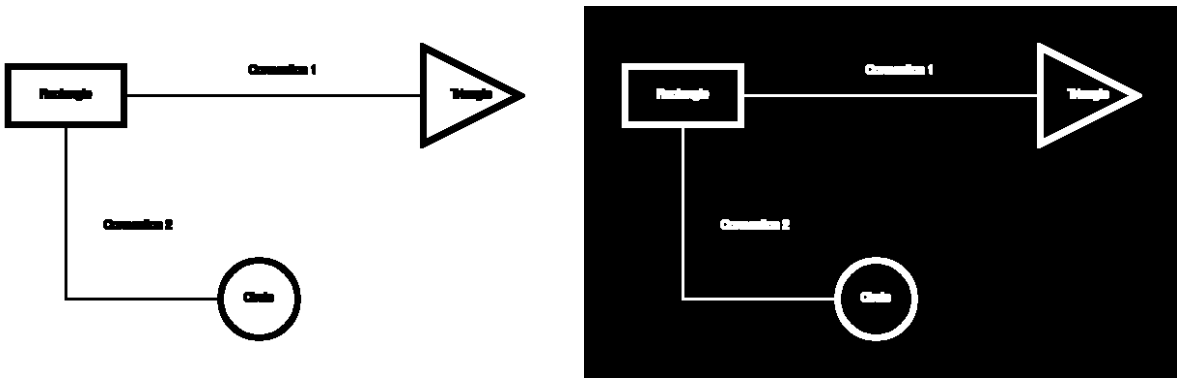
```
    def __prepare_image(self, img):
        img_grayscale = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        blurred = cv.GaussianBlur(img_grayscale, (3, 3), cv.BORDER_DEFAULT)

        _, binary_img = cv.threshold(blurred, 250, 255, cv.THRESH_BINARY)
        inverted_binary_img = ~ binary_img

        kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, (6, 6), (-1, -1))
        nodes = cv.morphologyEx(inverted_binary_img, cv.MORPH_OPEN, kernel)

        nodes_larger = cv.morphologyEx(nodes, cv.MORPH_DILATE, kernel)
        links = cv.subtract(inverted_binary_img, nodes_larger)

        nodes_larger = cv.morphologyEx(nodes_larger, cv.MORPH_DILATE, kernel)
        intersections = cv.bitwise_and(nodes_larger, links)

        return (nodes, nodes_larger, links, intersections)
```

[5] URL to the forum topic: https://answers.opencv.org/question/101717/visually-anayzing-network-diagrams/

Listing 7 shows a method that receives an image. The image is first transformed into a grayscale image. Afterwards, Gaussian blur [29] is applied to the image. Gaussian blur is used to blur an image with the help of a Gaussian filter. It works by convolving each point that is part of the input array with the filter and then producing an output array by summing them all. The filter is calculated with the two dimensional Gaussian function [30, p. 393]:

$$\mathrm{f(x,y)} = \mathrm{A.e}^{-\left(\frac{(\mathrm{x}-\mathrm{x_0})^2}{2\sigma \mathrm{x}^2}+\frac{(\mathrm{x}-\mathrm{y_0})^2}{2\sigma \mathrm{y}^2}\right)} \tag{5.1}$$

In the equation, A represents the amplitude, (x_0, y_0) is the center, while σx and σy the deviations in x and y directions. In the context of the project, Gaussian blur was used to reduce noise created by using the thresholding function. This function is used to create a binary image [31], an image that only has black and white pixels, by providing a threshold value in the range from 0 to 255. If the pixel has a value greater than the threshold value, it is equal to the $maxValue$, in this case 255. Afterwards, an inverted image is created so the image has a black background with white elements on it, as computer vision works better with that kind of input. The results of those operations can be seen in figures 9a and 9b.



(a) Applying Gaussian blur (b) Invertion of the image

Figure 9: Results of the first part of preprocessing [Author's Work]

The next operation used is called *opening* [32] and it is actually a combination of two basic morphological transformation methods offered by OpenCV called *dilation* [33] and *erosion*. In short, these operations are used to clean up noise from images, isolation or joining elements and for finding holes in images or intensity bumps. In both operations, a kernel of a certain shape, like a circle or square is applied over the image. Applying dilation to an image results in brighter spots on the image growing, while using erosion causes the brighter spots to shrink. The opening operation applies erosion first, and afterwards dilation. In context of diagrams, this causes the links between shapes, or nodes to disappear. Subsequently, to extract links between the nodes, the extracted nodes are subtracted from the complete image and the result can be seen in figure 10a. Dilation is once more applied to the image so the size of shapes are more pronounced. With that, by using the "bitwise and" operation on the enlarged extracted nodes and on the extracted links, intersections between nodes and links are obtained. Intersections are needed to calculate which nodes are connected to which links. Figure 10b presents the intersections extracted from the example input image. The preprocessing method returns

a tuple with the nodes, enlarged nodes, links and intersections so these images can be used further down the line in the algorithm.

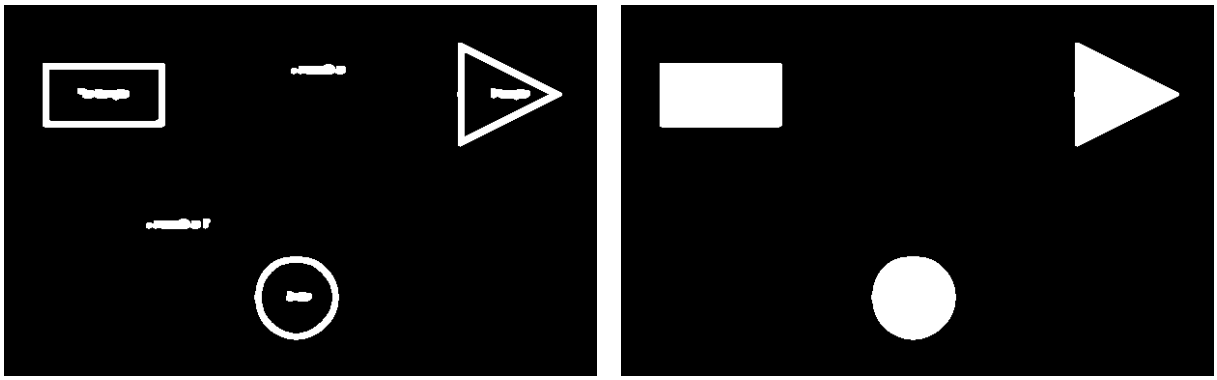
(a) Extracted connections (b) Extracted intersections

Figure 10: Results of diagram data extraction by subtraction and bitwise and function [Author's work]

It should be also mentioned that during testing of the diagram interpreter, a bug was noticed that causes the interpreter to assign a unary relationship to a certain vertex. After some research, it was noticed that the bug was caused after using Gaussian blur. As a matter of fact, those vertexes that contain a label can be falsely identified as having unary relationships because the blurring of the image may make the text to appear as a blob. That blob could be, in later steps of preprocessing phase, identified as an intersection that appears in the area of vertex, and thus assumed that the vertex has a connection to itself. This problem can be fixed in two ways - by forcing the algorithm to ignore the area where the label was detected, or by filling the contour so the text is hidden. The latter approach was chosen and by utilizing a method called fillPoly that fills any polygon with a color of choosing. Firstly, the contours around the image are found. After that, the list of contours is iterated through and the image is filled with colours according to the size of the contour. Specifically, if the contour is large, it is considered a vertex and it is filled with white colour, otherwise it is coloured black so it does not interfere with the algorithm. The content of listing 8 should be inserted between the 9th and 11th line of code of listing 7. How the input image was affected by this change can be seen in figures 11a and 11b.

Listing 8: Using fillPoly to make image more readable

```
1         vertexes, _ = cv.findContours(nodes, cv.RETR_EXTERNAL, cv.
           CHAIN_APPROX_SIMPLE)
2         for vertex_num in range(len(vertexes)):
3             selected_contour = vertexes[vertex_num]
4
5             if (cv.contourArea(selected_contour) > 1000):
6                 cv.fillPoly(nodes, pts =[selected_contour], color=(255,255,255))
7             else:
8                 cv.fillPoly(nodes, pts =[selected_contour], color=(0,0,0))
```



(a) Before using fillPoly (b) After using fillPoly

Figure 11: Comparison of the detected nodes before and after using the fillPoly operation [Author's work]

5.3.3. Detecting Shapes

It can be seen in listing 9 that the method takes a list of contours detected from the preprocessed images as a parameter. Those contours are extracted from the images by calling a method provided by OpenCV called findContours. The contours are expressed as an array of coordinates that represent the outline (contour) of a detected object. The other parameter is an image that contains the extracted nodes. Firstly, endpoints are extracted from each contour and since they are encapsulated inside an array, they are extracted from it. By using the cv.moments method, center coordinates from the shape are extracted. boundingRect method can be used to draw a rectangle around a contour, but in this instance it is used to get coordinates of the contour and height and width of the contour.

Listing 9: Coordinate extraction

```
1     def __detect_shapes(self, vertexes, nodes):
2         shapes = []
3         inverted_node_image = ~nodes
4         for vertex_num in range(len(vertexes)):
5             selected_contour = vertexes[vertex_num]
6             end_points = (cv.approxPolyDP(selected_contour, 0.01 *
7                           cv.arcLength(selected_contour, True), True)).tolist()
8             cleaned_end_points = []
9             for ep in end_points:
10                cleaned_end_points.append(ep[0])
11            M = cv.moments(selected_contour)
12            cX = int(M["m10"] / M["m00"])
13            cY = int(M["m01"] / M["m00"])
14
15            x, y, width, height = cv.boundingRect(selected_contour)
```

The next part of diagram analysis is a small one, however it is crucial for the application: object recognition. It is an essential part of computer vision and it works by taking an image as an input and returns an output value that represents the classification of the object. Throughout the development of the application, object recognition was accomplished through three differ-

ent approaches. The first approach, object recognition based on vertexes, was the most simple one. The number of end points determined what kind of geometrical shape the contour represented; three endpoints meant it was a triangle, four endpoints a rectangle, etc. This idea was implemented using the Chain of Responsibility [34] design pattern. For each shape a handler exists that checks if the number of end points aligns with that of a certain geometrical shape. On condition that the number of end points is equal to the required number, a new Shape object is created. Otherwise, the request is forwarded to another ShapeHandler. Listing 10 contains the implementation of RectangleHandler class.

Listing 10: Rectangle shape handler

```
1     class RectangleHandler(AbstractShapeHandler):
2     def handle_shape(self, id, end_points: Any) -> Optional[str]:
3         if len(end_points) == 4:
4             return Shape(id, "Rectangle", end_points)
5         else:
6             return super().handle_shape(id, end_points)
```

The first problem with that approach is unreliability, as there are many shapes with the same amount of end points. Furthermore, if the shape is not properly preprocessed, it could return a wrong number of end points. Secondly, it is impossible to properly classify a circle, because the number of end points is always unpredictable.

The second approach to object recognition was OpenCV provided method matchShapes. This method compares two images and tries to determine how similar the shapes inside the images are. More precisely, the similarity is established by calculating Hu invariants. That means that the shape can be scaled, rotated and reflected and still be recognized as the same shape to the original object. The closer the output value to 0, the bigger the likelihood of the shapes being identical. Object recognition with this method worked by creating an image repository of all the shapes that would be used in the application. Then, all of the images would be loaded into the application upon the startup of the server. Inside the object recognition method, after it was determined that a contour was larger than a certain area, the matchShapes method is called. As shown in listing 11, each of the shapes inside the repository is checked against the given contour. As a result, the minimum output value of the method and the shape it was calculated with are saved.

Listing 11: Shape recognition with help of matchShapes method

```
1     if (cv.contourArea(selected_contour) > 1000):
2         lowest_rating = 1
3         for shape in self.shape_repository:
4             ret = cv.matchShapes(selected_contour,shape[1],1,0.0)
5             if(ret<lowest_rating):
6                 lowest_rating = ret
7                 selected_shape = shape
```

While this approach was more precise and reliable, it was observed that certain shapes are wrongly classified. For instance, triangles would be classified as circles. With further experimentation, it was deduced that the matchShapes method is not able to correctly classify

shapes that have been stretched. Continuing with the triangle example, if the shape repository contained an equilateral triangle, while the contour was that of an orthogonal triangle, then the matching method would not recognize them as the same shape.

After analyzing the possible solutions to this problem, only two of them were viable: extension of the shape repository with images that cover various types each shape can be or implementation of a machine learning model. It was decided that using a machine learning model was the better option. The machine learning model was realized with a convolutional neural network. How the model was implemented and structured can be found in subsection 5.3.4. After creating the model, it was loaded into the DiagramInterpreter class during initialization. For the model to be able to classify the shape, it must first be extracted as an image from the diagram, since the input data can't be that of a contour. There is also an edge case that had to be taken into account when extracting the shape. The diagram can be created in black and white colour. Once the diagram was turned into a binary image, the shapes would be "hollow", while the images in the training dataset were "full". To fill out the hollow shapes, a OpenCV method called fillPoly was used. It uses contour coordinates to fill an area with colour.

Earlier in the object recognition method, coordinates and dimensions of the bounding rectangle were obtained. They are now used to isolate the object of interest into a new image with a slight margin around the shape, as shown in listing 12. This margin is necessary as the model was trained with a dataset that contains margins. Secondly, the model was trained with images with dimensions 100×100 pixels. That means the image must also be rescaled to this dimension. Furthermore, the input data has values in the range from 0 to 255, which indicates that the data must be normalized so the values fall inside the $[0, 1]$ range.

Figure 12: The input images that the machine learning model receives [Author's work]

Once the input data is transformed into an adequate format, it can be inserted into the machine learning model. The model predicts the output and offers a list of predictions that are expressed as probabilities with values close to 0 and close to 1 being considered least likely and most likely to be of a certain class, respectively. Those values are rounded up, and the index of the class where the value is equal to 1 is extracted from the list. Finally, the name of the class is fetched from a fixed list of classes with the help of this index. With the shape classified, a new Shape object is initialized and added to the list of shapes which the method returns.

Listing 12: Object extraction from the diagram and machine learning prediction

```
        if (cv.contourArea(selected_contour) > 1000):
                ROI = inverted_node_image[y-20:y+height+20, x-20:x+width+20]

                rescaled_image = cv.resize(ROI,(specified_width, specified_height),
                 interpolation=cv.INTER_LINEAR)

                image_array = keras.utils.img_to_array(rescaled_image)
                image_array = np.round(image_array/255,3).copy()

                image_array = np.expand_dims(image_array, axis = 0)

                prediction = self.cnn_model.predict(image_array)
                list_of_predictions = list(np.round(prediction[0]))
                predicted_shape = CATEGORIES[list_of_predictions.index(1)]
                new_shape = Shape(vertex_num, predicted_shape, cleaned_end_points, (
                 cX,cY), width, height)
                shapes.append(new_shape)
        return shapes
```

5.3.4. Implementation of the Convolutional Neural Network

In section 3.4 it is explained what convolutional neural networks are, how they work and how they are structured. This subsection brings that knowledge into practice and explains how the machine learning model was created. The network was implemented using Keras [35], a popular machine learning API that is built on top of Tenserflow2. Keras was developed with the aim of making machine learning accessible to a broader group of users, simplifying the whole process of developing a model. Before a model could be made, it was crucial to find a dataset which contained the basic shapes needed for classification: circle, triangle and rectangle. On Kaggle, a dataset that mostly fit the description was found [6]. The dataset contains 16,000 images of circles, squares, triangles and stars. Since the stars are not needed for this project, they can be ignored, but the problem was that the dataset did not contain rectangles, but only squares. There was no guarantee that the model would be able to correctly classify rectangles, however this was still the biggest dataset that resembled the data used in this prototype, so it was decided that it would be used.

The first attempt at creating a model resulted in the model being only 32% accurate and no amount of network structure adjustment fixed the problem. Kaggle, besides offering access to a large amount of datasets, also gives users an opportunity to share their solutions to getting the best possible model. It was decided that the implementation of the machine learning model would be made following the steps of such a solution that claimed it had a 100% accuracy [7].

Listing 13: Data preparation for the machine learning model

```
categories = ['circle', 'square', 'star', 'triangle']
```

[6] The URL for the dataset: `https://www.kaggle.com/datasets/smeschke/four-shapes`

[7] The URL to the solution: `https://www.kaggle.com/code/victorbnnt/100-accuracy-shape-recognition`

```
for cat in categories:
    filelist = glob.glob('./shapes/' + cat + '/*.png')
    target.extend([cat for _ in filelist])
    data.extend([np.array(Image.open(fname).resize((im_width, im_height))) for fname
      in filelist])

data_array = np.stack(data, axis=0)
X_train, X_test, y_train, y_test = train_test_split(data_array, np.array(target),
 test_size=0.2, stratify=target)

X_train_norm = np.round((X_train/255), 3).copy()
X_test_norm = np.round((X_test/255), 3).copy()

encoder = LabelEncoder().fit(y_train)
y_train_categories = encoder.transform(y_train)
y_test_categories = encoder.transform(y_test)

y_train_one_hot_coded = to_categorical(y_train_cat)
y_test_one_hot_coded = to_categorical(y_test_cat)

X_train_norm = X_train_norm.reshape(-1, 100, 100, 1)
X_test_norm = X_test_norm.reshape(-1, 100, 100, 1)
```

Listing 13 shows how the dataset was loaded into the application's memory. The dataset is segmented into folders which bear the names of the classes which the shapes belong to. First, by cycling through categories, each image that is inside a category's specific folder is loaded into an array. Additionaly, the images are rescaled to dimension 100×100 to get better performance. An additional axis is added to the data and afterwards using train_test_split, the data is split into training and testing data, where tesiting data makes up 20% of the whole dataset. The data is grouped into input values and labels with X and y, respectively. Just like with the input image that the complete model receives, the dataset input values need to be normalized. That is why they are divided by 255 to get values in the $[0, 1]$ range.

The data labels also need to be transformed into data understandable to computers. This is achieved using LabelEncoder[8], a method provided by a Python library called skicit-learn. It is used to transform textual categories into numerical ones (triangle = 0, circle = 1, ...). The categories are then transformed into vectors with their size equal to the number of possible classes, i.e [1,0,0,0] represents a triangle. That process is called one-hot encoding [36]. Lastly, the Keras algorithm must receive four-dimensional data, with the fourth dimension being the number of channels. The images are grayscale, which means the number of channels is one. In case, that information is not specified, Keras assumes that the number of channels is three. By using the reshape method, it is clearly stated that the images are grayscale.

Listing 14: Structure of the convolutional neural network

```
def initialize_model():
    model = Sequential()
```

[8] LabelEncoder URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

```
3       model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(im_height,
         im_width, 1), padding='same'))
4       model.add(layers.MaxPool2D(pool_size=(2,2)))
5
6       model.add(layers.Conv2D(64, (3,3), activation='relu', padding='same'))
7       model.add(layers.MaxPool2D(pool_size=(2,2)))
8
9       model.add(layers.Conv2D(128, (3,3), activation='relu', padding='same'))
10      model.add(layers.MaxPool2D(pool_size=(3,3)))
11
12      model.add(layers.Flatten())
13      model.add(layers.Dense(120,activation='relu'))
14      model.add(layers.Dense(60,activation='relu'))
15      model.add(layers.Dropout(rate=0.5))
16
17      model.add(layers.Dense(4, activation='softmax'))
18
19      return model
```

As it is known, convolutional neural networks are made of many layers. The most common sequence of repeating layers being a convolutional layer, followed by an activation layer, a pooling layer. After a number of convolutional layers, it is normal to add a few fully-connected layers with ReLU activation, a dropout layer and finally a fully-connected layer with Softmax activation and the number of neurons equal to the number of expected classes.

Listing 14 shows that this network begins with a convolutional layer of 32 neurons and a 3×3 kernel. The first layer must also specify the shape of the input data, in this case $(100, 100, 1)$. The layer is followed by an ReLU activation layer and padding set to "same" which according to the official documentation [37], means that zeroes are evenly padded around the input from all directions. Every following convolutional layer has double the neurons as the one before. The first two pooling layers have receptive fields sizes 2×2 with stride being adapted to the field size. The third layer has the kernel size set to 3×3 and then the network is flattened. Followed by that operation is a fully-connected layer with ReLU activation and 120 neurons. Another fully-connected layer is added to the network, however it has 60 neurons. Just before adding the final fully-connected layer, the dropout operation with the rate of 0.5 is used to make the model more robust.

Listing 15: Compilation of the model

```
1 model = initialize_model()
2 model.compile(optimizer='adam', loss='categorical_crossentropy', metrics='accuracy')
3 es = EarlyStopping(patience=5, monitor='val_accuracy', restore_best_weights=True)
4
5 history = model.fit(X_train_norm, y_train_oh, batch_size=16, epochs=1000,
   validation_data=(X_test_norm, y_test_oh), callbacks=[es])
6
7 model.save('computer_vision/cnn_model.h5')
```

The last part of creating a convolutional neural network is initializing, compiling and training the model. As seen in listing 15, the model is initialized by calling the initialize_model

method introduced in listing 14. By compiling the model, the parameters for training are set and the model is ready for training. It must be taken into consideration that the training process can last a while, and after some iterations the progress rate could diminish. This is prevented by using EarlyStopping, a callback method that can monitor a value during training and if the value has not changed significantly over a certain number of training iterations, the training process stops early. The monitored metric here is "val_accuracy" and it stands for accuracy of the model during validation. Keras documentation [38] defines accuracy as a calculation of how often the predicted value equals the label:

$$accuracy = \frac{y_{predicted}}{y_{true}} \tag{5.2}$$

Another metric that is an important indicator of the quality of the trained model is loss. Since there are multiple classes, categorical crossentropy is used as loss function[39]. Crossentropy loss is calculated by comparing random expected values t with predicted values y:

$$crossentropy \quad loss = L(y,t) = -\sum_i t_i \ln y_i \tag{5.3}$$

The lower this metric, the more accurate the model. Finally, the model is trained with input batches the size of 16 for 1000 epochs or iterations. The trained model was evaluated and the accuracy was around 99%. Using the model to predict the shapes that would be used in diagrams showed that the model accurately predicted the category of the shape. Additionally, it was noticed that the model was not overfitted as it could accurately classify rectangles, even thought only squares were used in the training dataset. For the model to be used in the API, it was saved using the save method.

5.3.5. Finding the Connections

Another step in recreation of the diagram in the editor is finding which nodes are mutually connected. Looking back to the preprocessing step in subsection 6.3.2, the method returned intersections, links and enlarged vertexes of the diagram; this information can be used to find connections between the vertexes.

Listing 16: Connecting edges and vertexes

```
1         for intersection in intersection_list:
2             edge_index = ''
3             vertex_index = ''
4             for n in range(len(enlarged_vertex_list)):
5                 if(any(cv.pointPolygonTest(enlarged_vertex_list[n], (int(point
                   [0][0]),int(point[0][1])) , False)  >= 0 for point in intersection)
                   ):
6                     vertex_index = n
7
8             for e in range(len(edge_list)):
9                 if(any(cv.pointPolygonTest(edge_list[e], (int(point[0][0]),int(point
                   [0][1])) , False)  >= 0 for point in intersection)):
```

```
                    edge_index= e

            vertex_edge_list.append((vertex_index, edge_index))
```

Listing 16 shows a section of the method that is used to find the connections. The idea is straightforward - all of the data concerning the links (connections) is presented as coordinates and that data can be used to identify the overlaps in the data. If the diagram is correctly designed, every connection joins exactly two nodes. Going from that, every intersection corresponds to strictly one node and one connection. Conveniently, OpenCV offers a method called pointPolygonTest, which checks if a point is inside a given polygon. By iterating through the list of intersections, it is checked if any point belonging to the intersection area is inside some edge and vertex. Those vertexes and edges that contain the intersection are paired together and added to a list of tuples.

Listing 17: Grouping vertexes together

```
        for ve in vertex_edge_list:
            grouped_edges[ve[1]].append(ve)

        for edge in grouped_edges:
            connection = []
            for vertex in grouped_edges[edge]:
                connection.append(vertex[0])
            if len(connection) == 2:
                connected_vertex_list.append(connection)
```

Once the list of vertex and edge pairs is populated, it is possible to find which vertexes are connected to the same connection. As seen in listing 17, the edge-vertex pairs are grouped by the edges. Afterwards, all of the vertexes are inserted into a list. With the condition that the length of the list is equal to two, the list containing those two nodes is added to a list of connections.

Listing 18: Pairing connections and labels

```
          for shape in shapes:
                label_list = []
                for connection in connections:
                    if shape.id in connection:
                        if(connection[0] == shape.id):
                            shape.connections.append(connection[1])
                        else:
                            shape.connections.append(connection[0])

                for label in detected_labels:
                    point = Point(label.x, label.y)
                    polygon = Polygon(shape.end_points)
                    if(polygon.contains(point)):
                        label_list.append(label.value)
                        inner_labels.append(label)
                shape.text = ' '.join(label_list)

            outer_labels = [x for x in detected_labels if not x in inner_labels]
```

At this point, all of the parts of the diagram are acquired, they just need to be put together and returned to the web client. Listing 18 begins by iterating through a list of identified shapes. The list of connections between two nodes is checked whether the id of a shape corresponds to one of the two shapes inside the list. Those shapes that are part of the connection with the selected shape are added to the property of Shape called connections.

The next part of the algorithm is determining which identified text represents the label of a shape. A Point object is initialized with the coordinates of a label and an instance of a Polygon is created using the selected shape's end points. The label that is inside the polygon that represents the shape is considered the shapes label. That label is added to a list of inner labels that are going to be used for ascertain which labels do not belong inside any shape. With the help of list comprehension, a Python feature that enables users to create new lists with a shortened for-loop.

Listing 19: Creating a response JSON

```
1     shapes_dictionary = list(map(lambda shape: shape.asdict(), shapes))
2     label_dictionary = list(map(lambda label: label.asdict(), outer_labels))
3
4     data = {'ShapeList': shapes_dictionary, 'OuterLabels': label_dictionary}
5     return json.dumps(data)
```

To send the data back to the client, it must be formatted in a way that the client can process the data. This is usually done by sending data in XML or in this case, JSON format. Both the Label and the Shape classes have a method called asdict() which returns the data as a dictionary, which is the format that JSON uses. By utilizing a lambda function, all the shapes and labels are mapped into a list of dictionaries. Those dictionaries are then returned as JSON data to the client with the help of json.dumps() method. Listing 19 depicts how the data is mapped as JSON.

6. Editor

The development progress and break-down of how the diagram editor works is explained in this chapter. With code examples, it is clarified how data is sent to the API, how it interprets the response and the steps needed to create a diagram. Furthermore, the chapter contains a section about the library which is used to construct the editor and its components.

6.1. Choosing the Editor Framework

The start of the development of the diagram editor started by inspecting which diagram editing libraries and frameworks exist. There were many libraries and framework so it was necessary to filter out those that did not meet the criteria. The criteria for choosing the library to be used in the solution was as follows:

- The library or framework must be free.
- The framework must provide a simple way of inserting objects into the environment.
- The objects must be capable of modification (rotation, stretching, scaling).
- There must be a way to programically generate objects.
- Objects inside the environment can be connected via non-directional links.
- The creation of the user interface should as be simple as possible.

By the end of the selection, the only viable library was mxGraph. This library is know for its use in draw.io, a web application used to generate diagrams that was the inspiration for this project. mxGraph is open source and available to download and use on Github. The library offers many utilities for creating a robust, yet flexible diagram creation environment. It is composed of a web client that handles the object drawing, and of a server module that is used to export diagrams. In the documentation, it is stated that the web client [40] is written in pure JavaScript, while the server module is available in PHP, Java and .NET. While the library and draw.io are still widely used, the development on the library has stopped in November of 2020[1]. With the aim to keep the web client as simple as possible, it was decided not to use the existing server module because it would require implementing interoperability since the AI was implemented in Python. Since there was no need to create a complex application, an example application [2] provided by the authors of mxGraph was modified for the purposes of this project.

[1]The information about the cease of development is available at: `https://github.com/jgraph/mxgraph`

[2]The example application is available at: `https://jgraph.github.io/mxgraph/javascript/examples/editors/diagrameditor.html`

6.2. Capabilites of mxGraph

As stated before, mxGraph is a robust library capable of great modification of the graphing environment. This section provides information about the parts of the library needed to create a diagram. The documentation provides a lot of information about how to create a whole editor step-by-step. To explain how the editor was made, the parts that make it up must be understood. This library is made of various prototypes that are used to implement an diagram editor. There is a lot of depth to the library, however the editor is not that important to the overall project, so this section will cover only the most important features of the library. The library is loaded into the application by providing the paths necessary for the application to run. The first path is that of the library which is assigned to the **mxBasePath** property. This property is used to load all of the libraries assets. The other parameters concern loading JavaScript files that are used in the application. The mxGraph API specification [41] states that the library is divided into eight packages which are automatically imported by the top-level **mxClient** class.

Although it may not seem so, one can create a graph without creating an editor. That is so because the **editor** [42] is a separate entity from the graph and is part of its own package that goes by the same name. The editor's main function is to provide users with the potential to perform actions that change the state of the graph. Essentially, it is an application wrapper for the graph which provides various useful elements and options such as: actions, dialogs, widgets (properties, toolbar, popup menu), templates and backend integration.

At the core of the mxGraph architecture [40] is the mxGraph model and its purpose is describing the structure of the graph. This component is provided by the **view** and **model** packages. The class which implements it is called mxGraphModel. Through this class, structural changes like inclusion of elements, modification, and removal of them happen. Furthermore, determining the structure of the graph, along with other capabilities such as grouping, styling and setting visibility of elements is offered by mxGraphModel API. Even though the model stores all changes made, the class that recieves the commands is mxGraph. The graph is made of objects of the mxCell class, a class that represents a cell that holds certain information. The appearance of those cells can be changed with mxCellRenderer by reading the properties of a certain cell defined in mxStylesheet.

A good editor is one that offers its users flexibility when making diagrams. One way of doing that is offering them various shapes that can be used to present data. Shapes are provided by the **shape** package and are subclasses of mxShape. Furthermore, mxGraph has the option of using event listeners and layout algorithms. These options are supplied by the **handler** and **layout** packages. Finally, two more packages need to be mentioned - **util** and **io** packages. The first package provides utility classes that make for a better user experience, such as copy-paste, drag-and-drop, constants, events, etc. The latter class is not utilized in this prototype, but it provides the app with the option to convert JavaScript objects into XML and vice versa.

6.3. Implementation of the Editor

As mentioned before, for the implementation of the editor, an already existing example available on the repository of mxGraph was used. It was so because the editor was not the focus of the project and this way time and resources would be saved. Most of the work that went into the editor was refactoring and modification of existing parts of the application. In figure 13, the original application called mxDraw can be seen.

Figure 13: Screenshot of the original application mxDraw [43]

The first change that occured was modification of the design and layout. The buttons of the application were enlarged, and so was the area of the graph. Another colour scheme was applied to the web page so the page looks more colourful. Moreover, buttons from the toolbar were modified so they only show elements relevant to this project. mxGraph has several configuration files that are used to set up various parameters and settings of the editor. The most important one being the configuration file of the editor. This file contains all the parameters that the editor uses like: templates, element styles, element behavior and toolbar. In this particular web client, the configuration is called *diagrameditor.xml*. The configuration file already contained all the templates used and along with styles and toolbar options.

A template behaves like a wrapper of a mxCell and is used to define an element that is allowed to be used inside the application. Some of the templates available in the application were: *container*, *text*, *rectangle*, *triangle*. In listing 20 the insertion of a triangle template is visible.

Listing 20: Adding a triangle template to the configuration

```
1 <Array as="templates">
2         <add as="triangle">
3                         <Shape label="Triangle" href="">
4                                 <mxCell vertex="1" style="triangle">
5                                         <mxGeometry as="geometry" width="70" height=
                                           "80"/>
6                                 </mxCell>
```

```
                        </Shape>
                </add>
</Array>
```

Defining the styles of the graph is possible by accessing the mxGraph class and inserting wanted stylesheets into the mxStylesheet property of the configuration. The number of style properties usable for an element depends on the element type. For instance, vertexes can have their fill color, font size, shape, alignment defined, but can not have end arrow and edge styles defined as those properties only concern edges that connect vertexes. An example of configuring a style for the default edge is presented in listing 21.

Listing 21: Configuration of the default edge's style

```
<add as="defaultEdge">
        <add as="shape" value="connector"/>
        <add as="fontSize" value="10"/>
        <add as="align" value="center"/>
        <add as="verticalAlign" value="middle"/>
        <add as="rounded" value="1"/>
        <add as="labelBackgroundColor" value="white"/>
        <add as="strokeColor" value="#36393D"/>
        <add as="strokeWidth" value="1"/>
        <add as="edgeStyle" value="elbowEdgeStyle"/>
</add>
```

The last part of the configuration are the default toolbar options. To add a toolbar option, the name of the option must be defined. Moreover, the template of the option must be defined, while the style is optional. Lastly, an icon of the option can be set by providing the path to a icon. The configuration of the default toolbar that is used in the application is shown in listing 22. From the listing, it is apparent that the toolbar has been stripped down so it provides the user only with rudimentary elements that can be recognized by the object recognition model: *text*, *rectangle*, *circle* (or ellipse) and *triangle*.

Listing 22: Configuration of the toolbar

```
<mxDefaultToolbar as="toolbar">
                <add as="Text" template="text" icon="images/text.gif"/>
                <add as="Rectangle" template="rectangle" icon="images/rectangle.gif"
                 />
                <add as="Ellipse" template="shape" style="ellipse" icon="images/
                 ellipse.gif"/>
                <add as="Triangle" template="actor" style="triangle" icon="images/
                 triangle.gif"/>
        </mxDefaultToolbar>
```

The final result of changing the style of the web client, and reconfiguration of the toolbar and editor styles and templates is visible in figure 14. The editor looks mostly the same except it is far more simple and the graph can cover an bigger area. The button to load an image is also included. That image is converted into a byte array, then put into the request body and sent as a POST request.

Figure 14: Screenshot of the modified diagram editor [Author's Work]

6.3.1. Communication with the API

A change event listener is added to the image input element. When the change in in the input element is detected, the loaded image is read using an instance of the FileReader class. Since the loaded image has some characters that are a problem when converting the data to JSON, they are removed using the replace method. The image data is then turned into JSON by using JSON.stringify() method. Afterwards, the request via Fetch [44], an API that is used to send requests and responses. For the request to be valid, parameters like headers, body and method must be defined. The first parameter is allowing CORS communication; by doing that, the request can be sent to a server outside the domain of the client. The second header parameter is explicitly defining the content type as JSON. Afterwards, the JSON is added to the body and the request is set to be sent as POST. The concrete example of how the request is sent to the API is shown in listing 23.

Listing 23: Preparation of the POST request

```
1   image_input.addEventListener("change", function () {
2     const reader = new FileReader();
3
4     reader.addEventListener("load", () => {
5       const uploaded_image = reader.result.replace("data:", "")
6         .replace(/^.+,/, "");
7       const json = JSON.stringify({
8         image: uploaded_image
9       })
10      const url = 'http://localhost:5000/image';
11
12      fetch(
13        url,
14        {
15          headers: { "Access-Control-Allow-Origin": "*" },
```

```
16            headers: { "Content-Type": "application/json" },
17            body: json,
18            method: "POST"
19          }
20        )
```

Once the data is returned by the API, the diagram can be constructed. As shown in listing 24, the diagram is constructed with a function called generateDiagram which takes the returned JSON as a parameter. The data is parsed and then the editor and model are fetched. The model is prepared for upcoming changes by calling the beginUpdate method. While the model is in update mode, it records all modifications made to the graph. To signalize that the update phase is over, the endUpdate is called. The first elements that are added to the diagram are vertexes. Using the shape of the detected object, a fitting template is obtained from the editor. If the template is valid, an instance of that template is cloned from the editor and assigned to that shape as it will be later used as a reference to connect to other vertexes. The object is added to the model and its geometry properties: coordinates, width and height, as well as the label are specified just as in the JSON.

Listing 24: Adding recognized shapes to the diagram

```
1     var parent = globalEditor.graph.getDefaultParent();
2     var model = globalEditor.graph.model;
3
4     model.beginUpdate();
5     try {
6       returnedData.ShapeList.forEach(shape => {
7         var template = globalEditor.templates[shape.shape.toLowerCase()];
8         if (template) {
9           shape.editorObject = model.cloneCell(template);
10          model.add(parent, shape.editorObject);
11          shape.editorObject.geometry = new mxGeometry(
12            shape.center_coordinates[0],
13            shape.center_coordinates[1],
14            shape.width,
15            shape.height);
16          shape.editorObject.setAttribute("label", shape.text)
17        }
18      });
```

Listing 25 shows that once the shapes have been placed on the graph, they are connected by iterating through each object's list of connections. A Shape object has a list of unique identifiers that represent other shapes. From the list of shapes an object that has the corresponding ID is acquired. If such an object exists, the two shapes are connected by using a mxGraph method called insertEdges(). The last part of diagram reverse engineering is adding the labels that are not part of any vertex. They are kept in a separate part of the JSON and they contain basic information like the coordinates and the value of the label. Those labels are placed onto the graph by cycling through the list of labels and just like with all other objects, the required template is fetched from the editor and the geometrical properties are set.

Listing 25: Adding connections and outer labels to the graph

```
1         returnedData.ShapeList.forEach(shape => {
2           shape.connections.forEach(connection => {
3             let otherVertex = returnedData.ShapeList.filter(e => e.id === connection)
                [0];
4             if (otherVertex) {
5               globalEditor.graph.insertEdge(parent, null, '', shape.editorObject,
                  otherVertex.editorObject, 'noArrowConnector');
6             }
7           });
8         });
```

Finally, the result of the application is presented in figure 15. To recall, the image was first loaded with the image loader function. Afterwards, it was sent via HTTP to the diagram interpreter API. The API preprocessed the image so OpenCV could work with it. Subsequently, the image was split into distinct segments: nodes, links, and intersections. Each node was extracted into a separate image that was sent to the machine learning model that analyzed the content of the image and returned a probability distribution of classes. Data from the links and intersections was paired with nodes they belonged to so the editor could programically apply those connections. Tesseract API was utilized to obtain text from the image. Ultimately, the data was parsed into JSON format and then returned via controller to the web client.

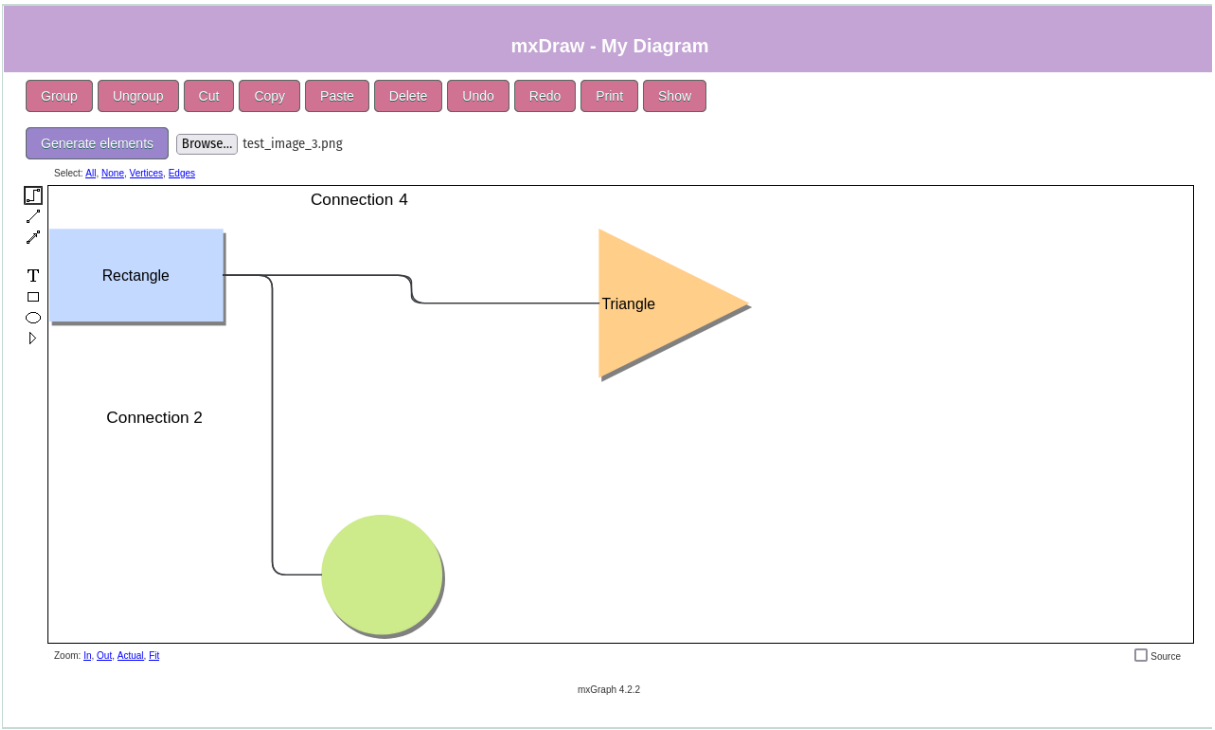


Figure 15: The final result of the application [Author's Work]

7. Conclusion

To realize this thesis, it was approached as an experimental research, where the experiment was the prototype web application. The technologies used for the implementation of the prototype consisted of Python and its libraries, Flask, HTML, OpenCV and mxGraph. The solution was developed in Visual Studio Code.

It was explained what artificial intelligence is, what it encapsulates and how it is related to machine learning. Computer vision was introduced as a subfield of artificial intelligence that focuses on simulating the way that humans visually perceive their surroundings. Neural networks were introduced as machines that are capable of learning through techniques such as backpropagation, weight and bias sharing. Additionally, perceptrons, building blocks of neural networks, were analyzed. Secondly, the structure and the algorithm of convolutional neural networks was investigated. It was noted that convolutional neural networks are the most fitting technology for visual object recognition. Furthermore, it was established that there are various real-life uses for computer vision in domains such as healthcare, military, entertainment and automotive industry.

The second part of the thesis started with the breakdown of the software architecture. The technologies used in development were explained along with the architecture diagram. The application was built upon the RESTful architecture which uses HTTP protocols to establish communication between the client and the server. Thereafter, the analysis of implementation of the web server was conducted. To make the inner workings of the server easier to follow, a use case scenario was created. The scenario consisted of an image that was sent by the client to be interpreted. The first part of the server that was observed was the controller. After that, the class called DiagramIntepreter was presented. The procedure of image interpretation starts with preprocessing. After the image was converted to the format acceptable to OpenCV, the object recognition starts. It was noted that the approach to object recognition changed throughout the development from simpler mathematical approaches to the utilization of a computer vision model. Finally, the elements of the diagram were brought together into a single JSON file that was sent as a response to the client.

The client was derived from an example provided on the documentation of mxGraph. It was developed in HTML and JavaScript and allowed the user to load an image that is sent to the server. The client is an diagram editor that provides the user with creating diagrams using three basic shapes: triangles, rectangles and circles. The connections between the shapes are non-directional only. The other condition while using the diagram is that between shapes can be only one connections. After the server sent the response, the diagram was reconstructed from the JSON.

Bibliography

[1] A. Trafton. “How the brain recognizes objects.” en. (10/2015), [Online]. Available: https://news.mit.edu/2015/how-brain-recognizes-objects-1005 (visited on 08/06/2022).

[2] V. Lakshmanan, M. Görner, and R. Gillard, *Practical machine learning for computer vision: end-to-end machine learning for images*, eng, First edition, first release. Bejing Boston Farnham: O’Reilly, 2021, ISBN: 9781098102364.

[3] B. Copeland. “Artificial intelligence | Definition, Examples, Types, Applications, Companies, & Facts | Britannica.” en. (03/2022), [Online]. Available: https://www.britannica.com/technology/artificial-intelligence (visited on 08/06/2022).

[4] T. M. Mitchell, *Machine Learning* (McGraw-Hill series in computer science). New York: McGraw-Hill, 1997, ISBN: 9780070428072.

[5] C. M. Bishop, *Pattern recognition and machine learning* (Information science and statistics). New York: Springer, 2006, ISBN: 9780387310732.

[6] T. G. Mesevage. “What Is Data Preprocessing & What Are The Steps Involved?” en. (05/2021), [Online]. Available: https://monkeylearn.com/blog/data-preprocessing/ (visited on 08/09/2022).

[7] S. García, S. Ramírez-Gallego, J. Luengo, J. M. Benítez, and F. Herrera, “Big data preprocessing: Methods and prospects,” *Big Data Analytics*, vol. 1, no. 1, p. 9, 11/2016, ISSN: 2058-6345. DOI: 10.1186/s41044-016-0014-0.

[8] A. D. Distante, Distante, and Wheeler, *Handbook of Image Processing and Computer Vision*, Undetermined. S.l.: Springer International Publishing, 2020, OCLC: 1181903180, ISBN: 9783030423773.

[9] S. Haykin, *Neural networks: a comprehensive foundation*, English. Delhi: Pearson Education, 1999, OCLC: 643435359, ISBN: 9788178083001.

[10] M. A. Nielsen. “Neural Networks and Deep Learning: Chapter 1.” en. (2015), [Online]. Available: http://neuralnetworksanddeeplearning.com/chap1.html (visited on 08/19/2022).

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning* (Adaptive computation and machine learning). Cambridge, Massachusetts: The MIT Press, 2016, ISBN: 9780262035613.

[12] M. A. Nielsen. “Neural Networks and Deep Learning: Chapter 6.” en. (2015), [Online]. Available: http://neuralnetworksanddeeplearning.com/chap6.html (visited on 08/19/2022).

[13] A. Rosebrock. “Convolutional Neural Networks (CNNs) and Layer Types.” en-US. (05/2021), [Online]. Available: https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/ (visited on 09/14/2022).

[14] IBM. “What is Computer Vision? | IBM.” en-us. (2022), [Online]. Available: https://www.ibm.com/topics/computer-vision (visited on 08/25/2022).

[15] Simplilearn. “What Is Computer Vision: Applications, Benefits and How to Learn It.” en-US. (04/2021), [Online]. Available: https://www.simplilearn.com/computer-vision-article (visited on 08/25/2022).

[16] D. Khemasuwan, J. S. Sorensen, and H. G. Colt, “Artificial intelligence in pulmonary medicine: Computer vision, predictive model and COVID-19,” en, *European Respiratory Review*, vol. 29, no. 157, p. 200 181, 09/2020, ISSN: 0905-9180, 1600-0617. DOI: 10.1183/16000617.0181-2020.

[17] A. Esteva, K. Chou, S. Yeung, *et al.*, “Deep learning-enabled medical computer vision,” en, *npj Digital Medicine*, vol. 4, no. 1, p. 5, 12/2021, ISSN: 2398-6352. DOI: 10.1038/s41746-020-00376-2.

[18] N. Milisavljević, “Comparison of three methods for shape recognition in the case of mine detection,” en, *Pattern Recognition Letters*, vol. 20, no. 11-13, pp. 1079–1083, 11/1999, ISSN: 01678655. DOI: 10.1016/S0167-8655(99)00074-4.

[19] P. Svenmarck, L. Luotsinen, M. Nilsson, and J. Schubert, “Possibilities and challenges for artificial intelligence in military applications,” in *Proceedings of the NATO Big Data and Artificial Intelligence for Military Decision Making Specialists’ Meeting*, 2018, pp. 1–16.

[20] IBM Cloud Education. “What is an Application Programming Interface (API).” en-us. (08/2022), [Online]. Available: https://www.ibm.com/cloud/learn/api (visited on 09/08/2022).

[21] MDN contributors. “HTTP | MDN.” en-US. (09/2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP (visited on 09/08/2022).

[22] L. Gupta. “What is REST.” en-US. (04/2022), [Online]. Available: https://restfulapi.net/ (visited on 08/30/2022).

[23] R. T. Fielding. “Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST).” (2000), [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visited on 09/14/2022).

[24] OpenCV. “About.” en-US. (2022), [Online]. Available: https://opencv.org/about/ (visited on 08/31/2022).

[25] Python Software Foundation. “General Python FAQ — Python 3.10.6 documentation.” (2022), [Online]. Available: https://docs.python.org/3/faq/general.html#what-is-python (visited on 08/31/2022).

[26] MDN contributors. “Cross-Origin Resource Sharing (CORS) - HTTP | MDN.” en-US. (09/2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 09/01/2022).

[27] A. Shvets. “Singleton.” en. (2022), [Online]. Available: https://refactoring.guru/design-patterns/singleton (visited on 09/02/2022).

[28] Python Software Foundation. “Glossary — Python 3.10.6 documentation.” (2022), [Online]. Available: https://docs.python.org/3/glossary.html#term-metaclass (visited on 09/02/2022).

[29] OpenCV. “OpenCV: Smoothing Images.” (09/2022), [Online]. Available: https://docs.opencv.org/3.4/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html (visited on 09/03/2022).

[30] E. S. Gedraite and M. Hadad, “Investigation on the effect of a gaussian blur in image filtering and segmentation,” in *Proceedings ELMAR-2011*, 2011, pp. 393–396.

[31] OpenCV. “OpenCV: Miscellaneous Image Transformations.” (09/2022), [Online]. Available: https://docs.opencv.org/3.4/d7/d1b/group__imgproc__misc.html (visited on 09/04/2022).

[32] OpenCV. “OpenCV: More Morphology Transformations.” (09/2022), [Online]. Available: https://docs.opencv.org/3.4/d3/dbe/tutorial_opening_closing_hats.html (visited on 09/04/2022).

[33] OpenCV. “OpenCV: Eroding and Dilating.” (09/2022), [Online]. Available: https://docs.opencv.org/3.4/db/df6/tutorial_erosion_dilatation.html (visited on 09/04/2022).

[34] A. Shvets. “Chain of Responsibility.” en. (2022), [Online]. Available: https://refactoring.guru/design-patterns/chain-of-responsibility (visited on 09/05/2022).

[35] Keras Team. “Keras: The Python deep learning API.” (2022), [Online]. Available: https://keras.io/ (visited on 09/06/2022).

[36] AskPython. “One hot encoding in Python - A Practical Approach - AskPython.” en-US. (11/2020), [Online]. Available: https://www.askpython.com/python/examples/one-hot-encoding (visited on 09/06/2022).

[37] Keras Team. “Keras documentation: Conv2D layer.” en. (2022), [Online]. Available: https://keras.io/api/layers/convolution_layers/convolution2d/ (visited on 09/06/2022).

[38] Keras Team. “Keras documentation: Accuracy metrics.” en. (2022), [Online]. Available: https://keras.io/api/metrics/accuracy_metrics/#accuracy-class (visited on 09/06/2022).

[39] 365 Team. “What Is Cross-Entropy Loss?” (08/2021), [Online]. Available: https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/ (visited on 09/06/2022).

[40] JGraph. “mxGraph 4.2.2.” (10/2020), [Online]. Available: https://jgraph.github.io/mxgraph/ (visited on 08/31/2022).

[41] JGraph. “API Specification.” (10/2020), [Online]. Available: https://jgraph.github.io/mxgraph/docs/js-api/files/index-txt.html (visited on 09/07/2022).

[42] JGraph. “mxEditor.” (10/2020), [Online]. Available: https://jgraph.github.io/mxgraph/docs/js-api/files/editor/mxEditor-js.html#mxEditor (visited on 09/07/2022).

[43] JGraph. “mxDraw Example.” (10/2020), [Online]. Available: https://jgraph.github.io/mxgraph/javascript/examples/editors/diagrameditor.html (visited on 09/14/2022).

[44] MDN contributors. “Fetch API - Web APIs | MDN.” en-US. (09/2022), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (visited on 09/07/2022).

List of Figures

List of Listings

Appendices