POLITECNICO DI TORINO

MASTER's Degree in Communications and Computer Engineering

MASTER's Degree Thesis

Automated Recognition of Annotations from Electrical Circuit Drawings

Supervisor

Prof. Stefano GRIVET TALOCIA

Candidate

Carmen R. MONCADA Q.

Academic Year 2024-2025

Automated Recognition of Annotations from Electrical Circuit Drawings

Carmen R. Moncada Q.

Abstract

Image recognition constitutes a key component of machine learning that enables intelligent systems to interpret visual information. Conventional optical character recognition (OCR) tools often encounter challenges with variations in font style, text orientation, language, and the presence of technical symbols. To address these challenges, a convolutional neural network (CNN) was implemented to identify characters individually within annotated labels of electrical circuit images. A post-processing function was integrated to reconstruct the detected characters, ensuring semantic coherence and compliance with standard circuit notation. The proposed method was then compared with a conventional OCR tool to evaluate accuracy and flexibility. Experimental results demonstrate that this approach enhances technical consistency and streamlines the automation of circuit label annotation in intelligent systems.

ACKNOWLEDGMENTS

to my parents, for their infinite love and unwavering belief in me. I hope one day to give back all that they have so generously given me.

Table of Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
DL	Deep Learning.
ML	Machine Learning.
CNN	Convolutional Neuronal Network.
API	Application Programming Interface.
TF	TensorFlow.
YOLO	You only look once.

Chapter 1

Introduction

The utilization of artificial intelligence (AI) and machine learning algorithms for the resolution of tangible issues has been adopted in numerous domains in recent years. The field of applications is wide-ranging, encompassing the use of predictive analytics for the detection of fraud or aberrant patterns within the banking and financial sectors. Additionally, it extends to the implementation of algorithms for the early detection of serious diseases and the analysis of medical images. In the domain of transport and logistics, it is used for the purpose of making forecasts within the supply chain, among other applications.

These techniques have also been applied in an educational context with the aim of creating an intelligent system capable of identifying and understanding electrical circuit diagrams based on images. The purpose of this project is to implement a cutting-edge web service for students and professors to use as an interactive tool for analyzing electrical circuits. This platform, 'autoCircuits' [**1**], incorporates a range of functionalities designed to automate the generation of electrical circuit theory problems, as shown in **Figure 1.1**.

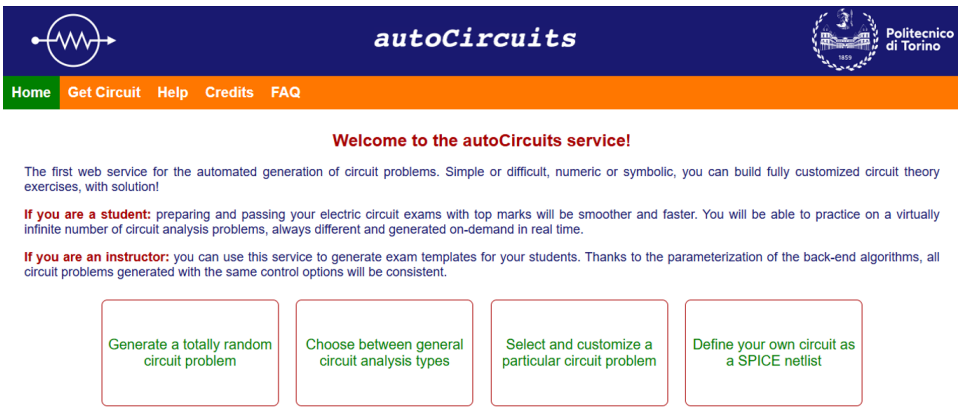


Figure 1.1: Web-service functionalities.

The primary objective is to assist students in enhancing their proficiency in circuit analysis and to provide a valuable support mechanism for professors by generating a vast number of unique circuit problems. This makes it an efficacious instrument for

creating consistent and diverse exam templates, as shown in **Figure 1.2**.

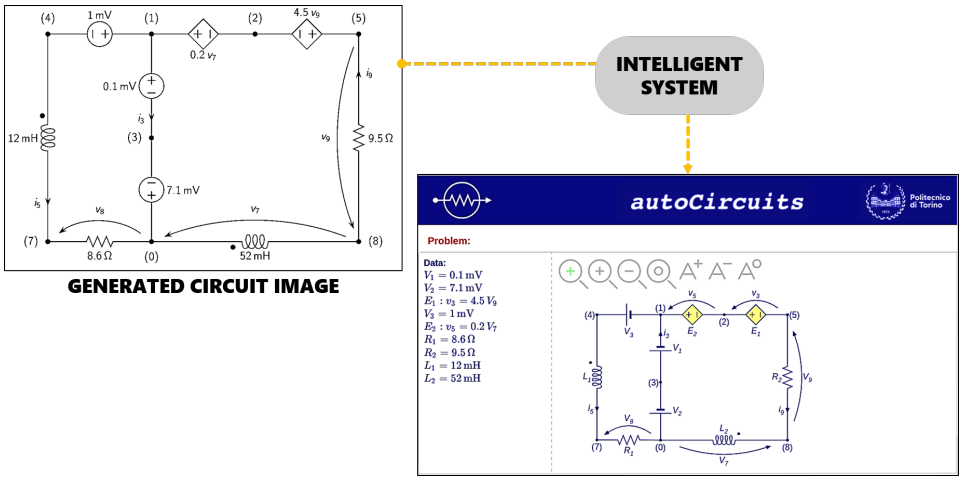


Figure 1.2: Electric circuit generated and analyzed by the system.

From the back-end perspective, the intelligent system process is initiated by the generation of an image containing the electrical circuit. This image is utilized as input to the trained YOLO (you only look once) model, which is a real-time object detection algorithm. All electrical components, outputs, and electrical elements (e.g., voltage and current arrows) along with their associated annotations, are detected in this stage. This is achieved through the recognition of standardized circuit symbols and the detection of junctions formed by the corresponding connection lines, as shown in **Figure 1.3**. Subsequently, all of these components are extracted from the circuit image so that they can be analyzed separately by means of image processing techniques, with a view to obtaining the complete topology of the network.

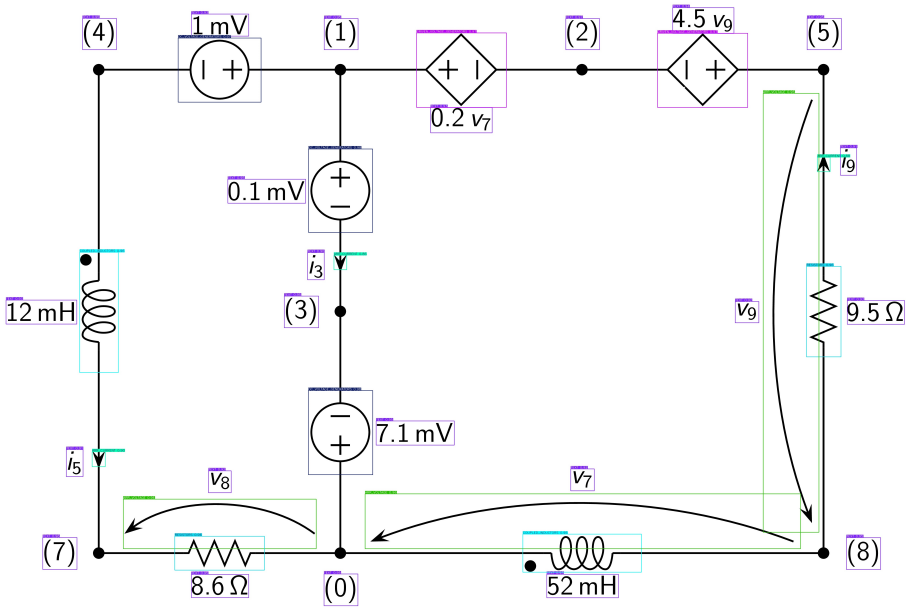


Figure 1.3: Circuits elements recognized by the intelligent system.

The data obtained by this method is incorporated into two nested structures, which are employed to reconstruct the original graph from the ground up. A comparison is finally made between the original and reconstructed graphs. This comparison is performed from two perspectives: graphical and structural, to evaluate the correct functioning of the system.

As mentioned before, the annotations within the image are extracted and independently detected through 'Pytesseract', an optical character recognition (OCR) tool, to recognize the labels associated with the electrical circuit. However, the issue arises due to Pytesseract's inability to accurately identify certain annotations present within the image, particularly those involving mathematical symbols or fractional formats for annotation. This method is heavily reliant on the quality of image resolution; consequently, low-resolution images, or those containing italic fonts for component annotations, present considerable challenges for accurate annotation detection and identification. The incorporation of multilingual labels further increases the complexity of the task.

These errors highlight the limitations of 'Pytesseract' in handling equation-style annotations, thus necessitating the use of an alternative tool capable of reliably identifying labels in electronic circuit components. Thus, the objective of this research is to explore two alternatives for accurately recognizing these annotations. For this purpose, the aim is to develop a function using a Convolutional neural network to detect characters and translate them into equation-style expressions, utilizing a post-processing function that assigns contextual meaning to each component. The performance of the proposed method is then evaluated by comparing it with another optical character recognition tool called Pix2Tex, which is predefined in the Python libraries.

This thesis is structured into seven chapters. Chapter 2 presents a comprehensive overview of machine learning algorithms, the principles underlying optical character recognition, and the fundamental components of a convolutional neural network.

Chapter 3 describes the construction of the dataset employed to train the proposed convolutional model, incorporating augmentation techniques to enhance data diversity.

Chapter 4 outlines the methodologies applied during the models training process, leading to the configuration that achieved the highest accuracy.

Chapter 5 of this text provides a comprehensive overview of the core function that facilitates the comprehensive process of recognizing annotations on a circuit. This function is employed to replicate the entire process on other computers.

Chapter 6 addresses the post-processing stage conducted after text extraction, detailing the methods used to assign contextual meaning to electronic circuit annotations in accordance with the models predictions.

Chapter 7 reports the experimental results obtained from evaluating the model on a substantial image dataset and provides a comparative analysis with the 'Pix2Tex' tool, a learning-based system designed to convert mathematical equations into LaTeX format. The results obtained from both methods, CNN and Pix2Tex, will be compared with the ground truth text annotations of the original image using a predefined function. This function reconstructs the circuit from the predictions generated by both models and compares the reconstructed structure with the original structure of the circuit.

Finally, the limitations of the proposed approach, including the challenges observed when applying both methods to new unseen data and the potential avenues for future improvement, are discussed at the conclusion, Chapter 8.

Chapter 2

State of Art

2.1 Machine Learning and Deep Learning

Machine learning is a branch of Artificial Intelligence that automates analytical model building. It is based on the idea that computers can learn automatically from data, identify patterns, make decisions autonomously, and improve performance and accuracy based on experience without being explicitly programmed [2, 3].

It comprises a set of algorithms and statistical models that enable computer or machine systems to carry out specific tasks without explicit instructions, relying instead on inference and patterns. On the other hand, Deep Learning, a subset of machine learning, improves the quality of learning environments by using multilayer artificial neural networks. These networks are designed to tackle complex tasks such as image recognition, natural language processing, and speech recognition.

The architecture of deep neural networks characteristically comprises several hidden layers, which are arranged within complex, deeply nested structures. Rather than relying on a single activation function, they frequently implement advanced operations, such as convolutions, or multiple activations within a neuron [4]. These capabilities enable deep neural networks to process raw input data directly and automatically learn the representations required for a given learning task. This ability constitutes the basis of the concept known as "deep learning".

Over the years, numerous deep learning architectures have been proposed to address a wide variety of learning tasks. Nevertheless, certain architectures demonstrate greater suitability for specific data types, such as time series, speech, or images. These distinctions arise primarily from differences in the complexity of the model, characterized by the types and numbers of layers employed, the number of neuronal units, and the structure of their interconnections.

2.2 Types of Deep Learning Architectures

The following section will present five deep learning architectures. These will be presented in rough order of development, with each successive model being developed to overcome a weakness identified in a previous model.

2.2.1 Convolutional Neuronal Networks (CNNs)

Convolutional neural networks (CNNs) are a specialized class of neural networks that are mostly applied in computer vision tasks such as image classification, object detection and pattern recognition. Exploiting principles from linear algebra, in particular matrix multiplication, CNNs automatically identify features and patterns in images and videos.

Structurally, CNNs consist of an input layer, several hidden layers, and an output layer, with nodes connected by weighted links and activation thresholds. The architectural design of the CNN will be explained in greater detail in the forthcoming sections. The main components include Convolutional layers, clustering layers, and fully connected (FC) layers. Through successive layers, they extract progressively more complex features: the first layers detect basic patterns such as edges or colors, while deeper layers capture higher-level structures, up to recognizing entire objects [4].

CNNs outperform traditional neural networks in handling high-dimensional data such as images, voice, and audio, eliminating the need for manual feature extraction. Their ability to share and process data between layers increases efficiency and mitigates overfitting. However, CNNs also present challenges: they are computationally intensive, require substantial resources such as GPUs [5, 6], and require expertise in tuning architectures and hyperparameters.

2.2.2 Recurrent Neuronal Networks (RNNs)

Recurrent Neural Networks (RNNs) are primarily used in natural language processing and speech recognition tasks, as they are designed to handle sequential or time-series data, event sequences, and natural language, enabling them to predict future outcomes.

The architecture of RNNs incorporates internal feedback loops, allowing them to learn sequential patterns and model temporal dependencies by forming a memory, using information from previous inputs to influence current inputs and outputs, as shown in **Figure 2.1**. They share parameters across layers and maintain consistent weight parameters within each layer, which are updated through backpropagation and gradient descent, supporting reinforcement learning [6].

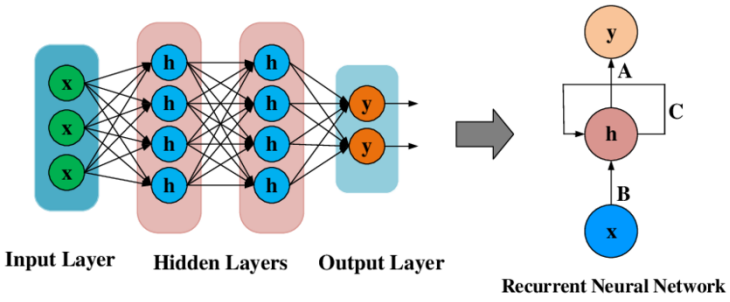


Figure 2.1: RNNs Architecture (Source:[7]).

Basic RNN architectures encounter difficulties stemming from vanishing or exploding gradients[5], which can greatly diminish or erase the influence of earlier inputs[2]. Both problems are associated with the gradient's magnitude, reflecting the slope of the loss function [6].

Vanishing gradients occur when gradients become progressively smaller, eventually causing the weight updates to diminish to zero and halting the learning process. Conversely, exploding gradients happen when gradients grow excessively large, leading to unstable models where weights may become undefined (NaN). A common strategy to mitigate these problems is to reduce the number of hidden layers, thus simplifying the model [5].

Additionally, these networks often demand long training times, face challenges when handling large datasets, and become progressively more difficult to optimize as the number of layers and parameters increases.

2.2.3 Distributed Representation

Distributed representations are of significant importance in the domain of Natural Language Processing (NLP). These representations are characterized by the use of continuous vectors in high-dimensional spaces to denote data, which may be words or phrases. The concept of similarity and semantic meaning is captured by allowing an entity to be represented by a pattern of values across multiple dimensions.

For instance, in Natural Language Processing (NLP) tasks, words with similar meanings are represented by vectors located near each other within the embedding space [2]. This proximity is not arbitrary but emerges from the patterns of contextual usage learned during training. Word embeddings address the issue of sparsity inherent in traditional text representation methods, such as one-hot encoding and bag-of-words (BoW) models, while simultaneously preserving the semantic relationships between words [8]. As a result, words that frequently occur in similar contexts within a corpus are positioned closely in the vector space, facilitating more effective modeling of linguistic structures.

However, it is important to note that this approach is not without its challenges. The requirement of large amounts of data to learn meaningful representations is a significant challenge, and as such, the embeddings may not capture the true semantic relationships if an insufficient amount of data is available. Furthermore, distributed representations can require significant computational resources for learning, necessitating substantial processing power and memory, particularly when dealing with large datasets.

2.2.4 Autodecoders

Autoencoders function similarly to distributed representations, employing an architectural framework that is also utilized in modern large language models (LLMs). They consist of two primary components: an encoder and a decoder. A basic architecture is illustrated **Figure 2.2**. The encoder compresses the input data into a dense, abstract representation, positioning similar data points closer together within the latent space. The decoder then reconstructs the original input from this compressed form. Through this process, the network is encouraged to preserve essential information in the latent space while filtering out irrelevant noise [2, 5].

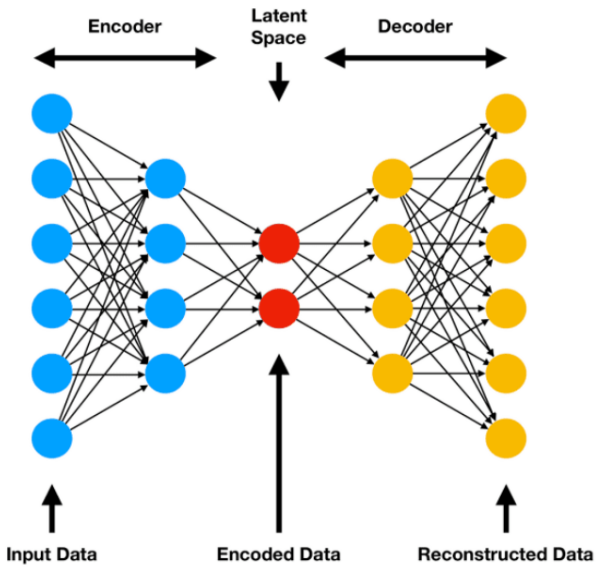


Figure 2.2: Basic Autoencoders Architecture (Source:[9]).

However, training deep architectures can be computationally intensive. During unsupervised learning, models may merely replicate input data rather than extract meaningful features[5]. Furthermore, autoencoders may struggle to identify complex relationships within structured data, potentially resulting in incomplete or inaccurate representations.

2.2.5 Generative Adversarial Neural Network (GAN)

Generative Adversarial Networks (GANs) are neural network architectures designed to generate new data samples that closely resemble the original training data. GANs consist of two main components: a generator and a discriminator, see **Figure 2.3**. The generator produces new samples, such as images, video, or audio, by learning the underlying distribution of the input data. The discriminator evaluates these generated samples by comparing them to real data, attempting to distinguish between authentic and synthetic inputs. Both networks are trained concurrently in a non-cooperative zero-sum game, where improvements in one network come at the expense of the other [2]. Training continues until the discriminator can no longer reliably distinguish between real and generated samples.

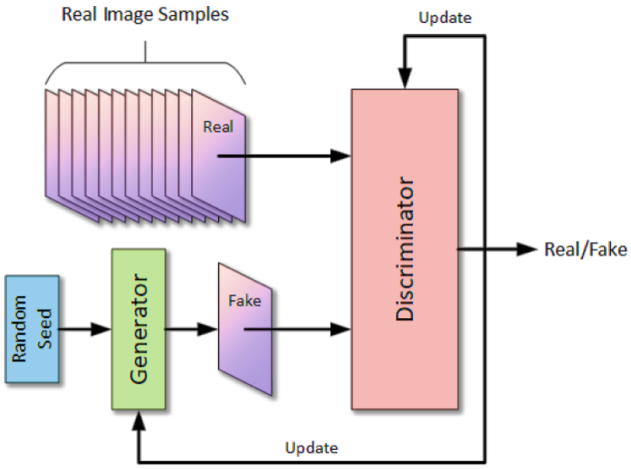


Figure 2.3: GANs Basic Architecture (Source:[10]).

GANs are effective in producing highly realistic data, which can be utilized to augment machine learning training processes, often requiring minimal or no labeled input. However, their training can be computationally demanding due to the prolonged adversarial dynamics between the generator and the discriminator, and typically requires large data sets to achieve satisfactory performance [5].

This work will be more in-depth in the area of image recognition, specifically in the area of optical character recognition.

2.3 Optical Character Recognition

Optical Character Recognition (OCR) is a technology that transforms images containing typed, handwritten, or machine-printed text into editable and machine-readable text. These images or documents can be classified into two groups, depending on the input mode: online and offline versions. Online recognition systems typically refer to images captured directly from a digital pen or tablet during the writing process. In contrast, offline recognition systems process text images or documents

obtained from digital cameras or scanners. It is acknowledged that the images in question may contain text produced by a machine, such as a printer or a typewriter, or alternatively by hand [11].

OCR systems contain eight key components: (i) collection of input images, (ii) input image preprocessing, (iii) text detection, (iv) character segmentation, (v) feature extraction, (vi) character recognition, (vii) handwriting recognition, and (viii) post-processing.

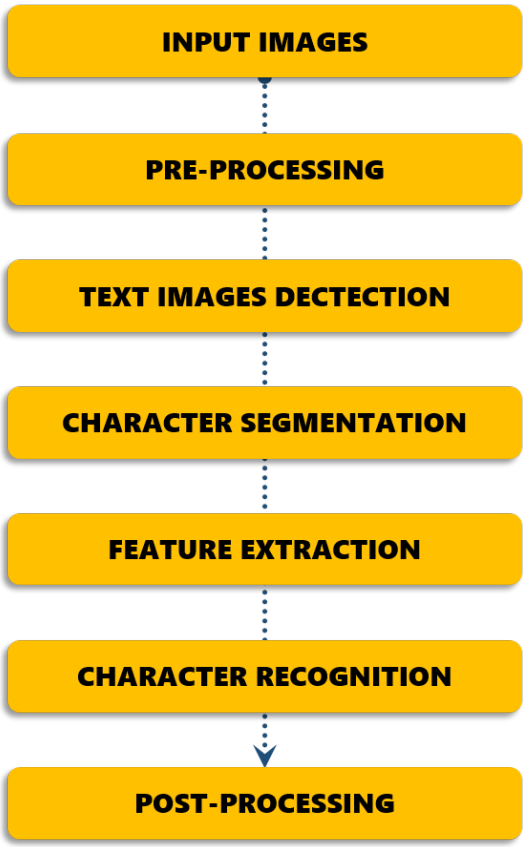


Figure 2.4: OCR components.

(i) **Collection of input images**: In this step, the dataset is created. It consists of acquiring handwritten or machine-printed documents and converting them into digital form. This process is performed using electronic devices such as digital cameras, scanners, telephones, or tablets to create the set of images.

In addition to the features and classifiers used, the dataset also plays a crucial role in determining recognition accuracy. Suppose that the input characters in the dataset are distorted, poorly written, or contain noise that cannot be effectively removed by pre-processing techniques. In that case, even the most advanced feature extraction or classification techniques will not be able to achieve the desired level of accuracy [11].

(ii) **Input image pre-processing**: Pre-processing is one of the most important steps in OCR. It involves a series of techniques applied to ensure that the quality of scanned document images is sufficient before the recognition phase begins. These techniques may include converting images to grayscale, thresholding, removing unwanted noise or artifacts, binarization, and deskewing, among others. The goal is to enhance image quality to achieve higher accuracy during the character detection phase. The specific techniques used in this work will be explained in detail in Chapter 3.

(iii) **Text area detection**: The text detection phase involves identifying text regions within images using machine learning models or edge detection techniques that can locate text areas even in complex layouts [12]. This is achieved by separating text from non-text elements such as graphs, backgrounds, and charts. This step is performed prior to segmentation to ensure that only the text areas are passed on to the subsequent processing stages.

(iv) **Character segmentation**: The segmentation process begins by dividing the text into individual paragraphs. Each paragraph is then segmented into lines, which are further broken down into words. These words are subsequently split into individual characters, and finally, each character may be decomposed into sub-characters for more granular analysis [11]. Segmentation techniques are often based on methods such as line detection, word separation, and advanced algorithms capable of distinguishing characters even in complex fonts or handwriting [12].

(v) **Feature extraction**: Feature extraction phase aims to identify and isolate the unique and distinguishing patterns of each character image, helping improve recognition accuracy by reducing the amount of data needed[12, 11]. These features can include lines, curves, corners, and pixel patterns that differentiate one character from another, allowing even similar-looking characters, such as "O" and "Q" or "1" and "I", to be correctly recognized.

(vi) **Character recognition**: Character classification constitutes the final phase in the OCR process. Following preprocessing, text area detection, and segmentation, each isolated character is analyzed and matched against a repository of reference patterns. Through this comparison, the system determines the most appropriate character label, thereby finalizing the recognition process.

(vii) **Postprocessing**: This constitutes the final stage following the character recognition phase. The primary objective of this function is to improve the accuracy of predictions during the classification process, ensuring alignment with the intended context and correcting any identified errors. Typically, this post-processing stage involves spelling correction, grammatical correction, and context-based adjustments to rectify misinterpretations made by the recognition model [12]. For instance, it may include correcting visually similar characters,

such as replacing "l" with "1" or "0" with "O", especially in cases where certain fonts render these characters nearly identically. Consequently, additional processing is necessary to refine and validate the predictions generated by the model.

2.4 Convolutional Neural Network Architecture Overview

As explained in the previous section, a convolutional neuronal network (CNN) is designed to mimic the human brain capabilities through the use of different layers of interconnected neurons. The architecture of a typical CNN is composed of four fundamental components: (a) a convolution layer, (b) a pooling layer, (c) an activation function, and (d) a fully connected layer. **Figure 2.5** depicts the organization of the layers in CNN.

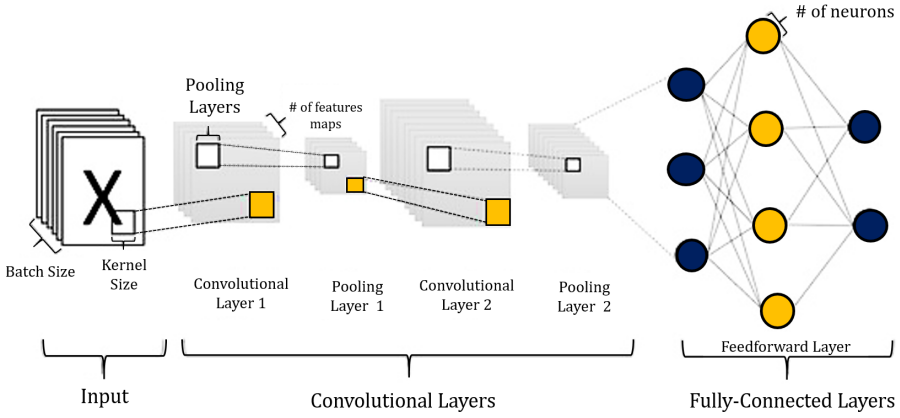


Figure 2.5: CNN layers architecture.

2.4.1 Convolutional Layer

It is composed of multiple learnable filters, also known as kernels, which are applied to the data before its use. The model performs convolutional operations on the input image. Each filter is characterized by its diminutive proportions, measured in terms of width and height, and its ability to seamlessly accommodate the input image. Within this framework, the filter performs the computation of dot products between its assigned weights and the respective input pixels [13, 14]. It typically follows a standardized numerical sequence, with filter sizes categorized as 3x3, 5x5, or 7x7. The operation is conducted to encompass the entire extent of the input volume (input image), and is subsequently followed by a nonlinear activation function (sigmoid, tanh, ReLU etc.).

The third dimension of the filter is analogous to the number of channels in the input. In the context of grayscale images, the depth value is set to 1. Conversely, color images are characterized by 3 RGB (Red, Green, Blue) color channels. This

process facilitates the identification of local patterns and features within the image. The output of the convolutional layer is a set of feature maps, which represent the presence of different features in the input image.

Finally, the output volume depends on three hyperparameters: depth, stride, and padding [13].

- The depth of the output volume is indicative of the number of filters used in the convolution operation. Each filter learns a distinct set of features from the input, including edges, blobs, and colors.
- The stride determines how many steps the filter slides in the input. When the stride is 1, the filters move one pixel at a time. When the stride is 2, the filters jump two pixels at a time as the filter is slid. This produces a smaller spatial output volume.
- Padding is used to influence the dimensions of the output in a convolution operation. Without padding, applying a convolution typically results in a smaller output, potentially discarding important data and details. To prevent this, extra zeros are added around the borders of the input. There are two frequently used padding methods: **valid** and **same**. **'Valid'** means no padding is added, so the output is smaller, while **'same'** ensures the output has the same dimensions as the input by adding the necessary padding.

2.4.2 Pooling Layer

This layer is employed after convolution layers to reduce the dimension of the feature maps (also referred to as sub-sampling or down-sampling). The application of the pooling operation to the input data is achieved by means of a filter that slides over it in the pooling layer (max, min, avg). The two most common types of pooling layers are max-pooling and average-pooling, where the maximum or average values are taken, respectively, when performing the sliding. However, Max-pooling is utilized more frequently than the average. Moreover, the hyperparameters of the pooling layer comprise the filter size and strides. The latter quantity denotes the number of pixels that the pooling window (or filter) shifts over the input feature map during each operation. To illustrate this, consider a scenario in which a stride of two pixels has been defined. In such a case, the pooling window will progress two units at a time after each operation [14], as illustrated in **Figure 2.6**. The pooling layer does not have parameters that can be learned.

2.4.3 Activation Function

In neural network architectures, each layer computes its output by performing a linear transformation on the output of the preceding layer. In the absence of

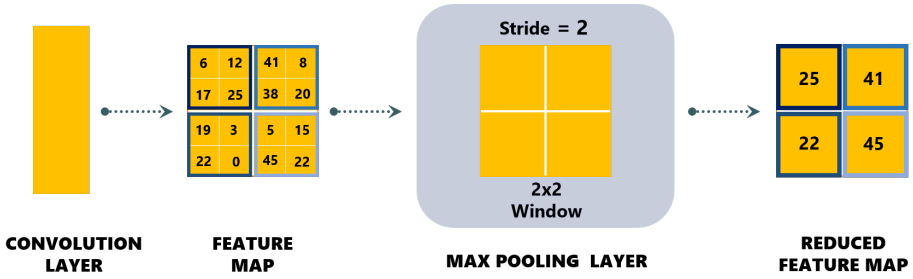


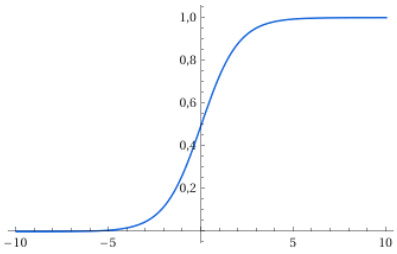
Figure 2.6: Max-Pooling Layer

any non-linear modification, this linear processing is propagated through successive layers, limiting the representational capacity of the model. However, introducing an activation function adds non-linearity to the model. This enables the network to approximate complex nonlinear functions, significantly enhancing its capacity to model diverse real-world patterns and relationships. An activation function is a mathematical function applied to the output of a filter. The most commonly used activation functions include the following:

2.4.4 Sigmoid

The sigmoid activation function maps input values to an output range between 0 and 1, making it suitable for tasks such as normalizing neuron outputs or modeling probabilistic predictions. It is particularly useful in binary classification problems, where the output can be interpreted as a probability [15, 16]. The mathematical formulation of the sigmoid function is given by:

$$f(x) = \frac{1}{1 + e^{-x}} \qquad (2.1)$$

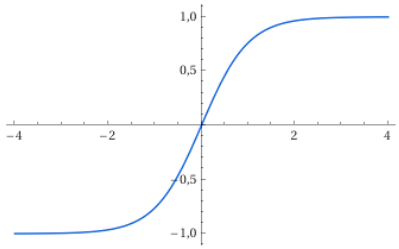


Despite its usefulness, the sigmoid activation function presents certain limitations. When a neuron's output approaches the extremes of 0 or 1, the derivative of the sigmoid function becomes very small [15]. As a result, during backpropagation, the gradients associated with these neurons tend to vanish, leading to minimal or no weight updates. This effect slows down learning and can propagate backward through the network, causing the gradients of earlier layers to diminish significantly, a phenomenon known as the vanishing gradient problem.

2.4.5 Tanh

The hyperbolic tangent (tanh) activation function transforms the input values into a range between -1 and 1, effectively centering the data and often leading to improved convergence during training compared to the sigmoid function, solving the problem of sigmoid functions not centering the output at 0 [15]. However, when the input values are very large or very small, the output of the tanh function saturates, resulting in small gradients. This saturation effect can hinder effective weight updates during backpropagation, contributing to the problem of vanishing gradients.

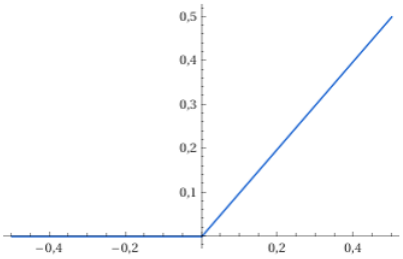
$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \qquad (2.2)$$



2.4.6 Rectifier Linear Unit (ReLU)

ReLU is one of the most widely used activation functions. It is a segmented linear function, specifically a ramp function, represented by the following equation:

$$f(x) = max(0, x) \qquad (2.3)$$



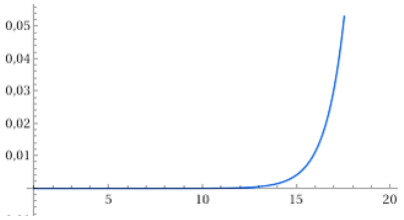
ReLU addresses several shortcomings of the sigmoid and tanh activation functions. Due to its piecewise-linear nature, ReLU is computationally efficient and faster to evaluate than its nonlinear counterparts. For positive input values, its derivative is equal to 1, which helps alleviate the vanishing gradient problem and facilitates faster convergence during gradient descent optimization [15].

However, it is important to note that this approach may be susceptible to the Dead ReLU problem. In essence, this problem occurs when the input is negative, resulting in a gradient that is precisely zero. Consequently, ReLU neurons are more prone to "dying" during the training process.

2.4.7 Softmax

The softmax function is a common activation function used in multi-class classification tasks. It is characterized by the transformation of the output scores into normalized probability distributions over multiple classes. The calculation of probabilities is achieved through the compression of the values of a real vector of length K between 0 and 1. To validate the probability distribution, it is necessary that the total sum of vector values is equal to 1. The outputs of the activation function are indicative of the estimated likelihood that the input belongs to each of the K categories. Higher values suggest a stronger confidence (probability) in the associated class [15]. This is characteristic of a K-class classification problem. The softmax function is defined by the following equation:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad \text{for } i = 1, \ldots, K \tag{2.4}$$



where x_i is the i-th element of the input vector x, and **K** is the total number of classes.

Moreover, a notable constraint of softmax functions emerges in scenarios where the input to the activation function attains a substantially negative value, leading to a gradient that approaches zero. Consequently, the corresponding weights receive minimal to no updates during the backpropagation process. The phenomenon of neuronal activation gives rise to neurons that remain inactive over time. These neurons are commonly referred to as "dead" neurons, and they fail to contribute to learning, which in turn affects the model's performance.

2.4.8 Fully-Connected Layer or Dense Layer

A fully connected layer is generally located at the end of the network for classification. Each neuron belonging to the fully connected layer is successively connected to all neurons and their preceding layers. It is customary for a CNN to process the input to generate multiple feature maps (after several convolution and pooling operations), which are then flattened into a single vector. The vector is then passed through one or more fully connected layers, which in turn transform spatial feature maps into class probabilities, and finally directs it to the output layer for classification. [13, 15].

2.4.9 Regularization

CNNs are typically used for image classification tasks, where overfitting is a significant concern. Overfitting occurs when the model generalizes or represents well

during training but performs poorly with new, unseen data in the test set. This occurs when the model has been intensively trained using all available information, but cannot generalize well with new information [17, 16]. This is where regularization techniques play a key role in the model, as they prevent overfitting during the training phase. The idea behind regularization is to increase the variability of the data at different stages of the CNN.

The regularization techniques are a set of algorithms that aim to reduce the error on data that does not belong to the training set. A prevalent regularization technique that is frequently employed in CNNs is Dropout.

2.4.10 Dropout

Dropout is a technique that facilitates regularization in the network by randomly removing some neurons or connections with a predetermined probability. This improves generalization by forcing the entire system to learn more features [15, 17]. The stochastic omission of specific connections or units leads to the formation of multiple sparse network architectures, from which a single representative model with reduced weights is selected. **Figure 2.7** illustrates the functionality of this method.

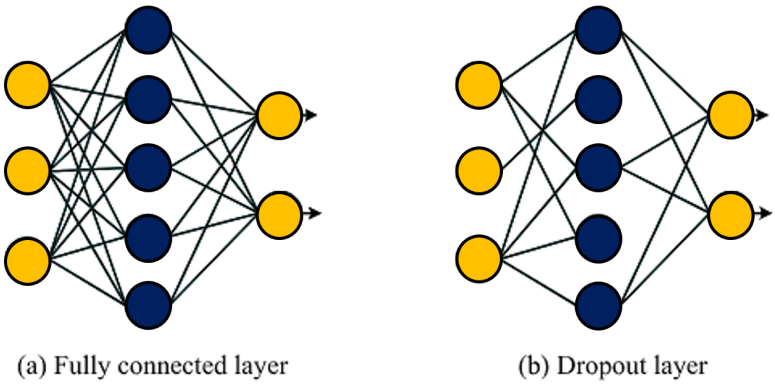


Figure 2.7: Comparison Between Fully Connected Layers and Dropout Layers

2.4.11 Cross Entropy Loss-Function

The loss function is an effective method to evaluate the divergence between the model's predicted outputs and the corresponding true values. This mechanism is of particular relevance in multi-class classification tasks, where class labels are required to be transformed into one-hot encoded vectors to facilitate comparison. Within this paradigm, the cross-entropy function calculates the loss for each class individually and subsequently adds them together to compute the overall loss. It quantifies the dissimilarity by the application of penalties to predictions that deviate from the true labels, thereby guiding the optimization process during model training. The equation

of the cross-entropy function is presented in 2.5.

$$\mathrm{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(x_i) \qquad (2.5)$$

N represents the total number of data samples, in this case, circuit images. y_i is the true probability of class i, and x_i is the predicted probability of class i.

2.5 Image Classification

In the domain of computer vision, image classification is widely regarded as one of the most challenging tasks. The objective is to differentiate between object classes, including but not limited to animals, vehicles, and people, based on the features presented in the images [15]. To perform image classification, it is necessary to select a classification model that is suited to the specific characteristics and requirements of the specified task.

The selection of a reference model for the character annotation recognition component of this thesis was largely based on the findings of Rizky (2023) [18], who conducted a comparative study of pre-trained CNN architectures for text recognition in images. The experiments demonstrated that applying transfer learning to standard CNN backbone architectures (e.g., VGG-16, ResNet18, DenseNet121, etc) and combining it with appropriate image augmentations (rotations, scaling, blurring) produces high accuracy, with VGG16 being the model that achieved high accuracy on the test set with 98.16%.

Since part of the task of this research work is the recognition of numerical and symbolic labels in circuit diagrams, it shares many characteristics with the character recognition scenario in [18] (small symbols, variable fonts, possible noise or distortions), and the same architectural options are promising. Therefore, it was decided to adopt VGG-16 as the base model, using transfer learning from ImageNet weights and employing some analogous augmentation strategies to make the model robust to variations in the quality of circuit annotation images. This choice is justified not only by the empirical performance described in their work, but also by the fact that their methodology addresses some of the challenges it is expected, such as different character sizes, rotations, variable fonts, and noise.

The subsequent section will provide a detailed exposition of the selected model and its architecture.

2.5.1 VGGNet Model

The VGG model is a convolutional neural network architecture developed by the Visual Geometry Group at the University of Oxford. The model was trained on the ImageNet dataset, which includes millions of annotated images across a wide range of object categories. This makes it well-suited to large-scale image recognition tasks. VGG-16 is distinguished by its simple and uniform architecture, a feature that has been identified as a contributing factor to its notable performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014. Despite its architectural simplicity, the model contains approximately 138 million parameters, thereby demonstrating significant representational capacity and computational demand.

Figure 2.8: VGG16 Model's Architecture (Source: [19])

The architecture of the model consists of 13 convolutional layers with 3x3 filters, a stride of 1, and the same padding, followed by 3 fully connected (dense) layers. The final dense layer is paired with a *Softmax activation* function for classification. Furthermore, the network incorporates five max-pooling layers, which serve to progressively reduce the spatial dimensions of feature maps. It is important to note that, although the model comprises 21 layers in total when accounting for all operations, as shown in **Figure 2.8**, only the 16 convolutional and dense layers possess trainable weights and are considered the weight-bearing components of the network.

Furthermore, there are OCR tools that facilitate the translation of images into LaTeX format. One of the most popular of these is called Pix2Latex. This OCR tool performs a similar task to that which is to be implemented in this work. The application receives an image containing a mathematical equation and translates it into its LaTeX representation.

In this thesis, the proposed method (CNN along with a postprocessing function) will be evaluated by means of a comparative analysis, in which the performance of the method is measured and contrasted with that of Pix2Tex. In the following section, an exposition will be made of the architecture and functionality of the aforementioned tool.

2.5.2 Pix2Tex

Pix2Tex is an open-source model developed by Lukas Blecher. It was designed to convert mathematical images into LaTeX markup. It employs an encoder-decoder framework, utilizing a scalable coarse-to-fine attention mechanism to generate presentational markup. The encoder typically consists of a Convolutional Neural Network (CNN), often based on a ResNet architecture, which extracts visual features from input images. These extracted features are then passed to a decoder, which is generally implemented using either Transformer or LSTM architectures, to sequentially predict LaTeX tokens. The entire process is illustrated in **Figure 2.9**.

The primary function of the encoder is to identify and represent the visual characters present in the input image as a set of feature embeddings. These embeddings are subsequently processed by the decoder to produce the corresponding LaTeX representation. Depending on the modeling approach, sequence prediction or classification, the training process may use either cross-entropy loss or Connectionist Temporal Classification (CTC) loss.

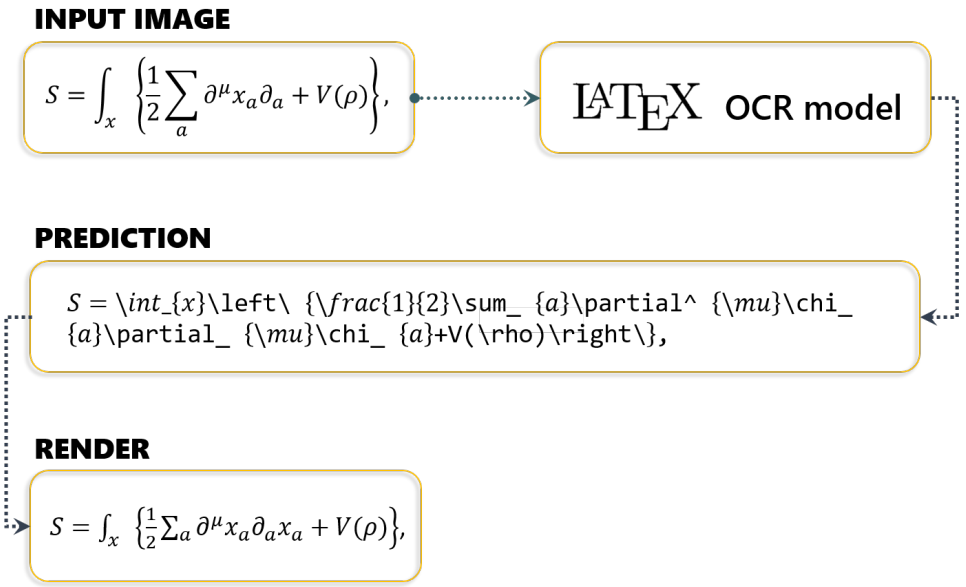


Figure 2.9: Pix2Tex process.

According to the documentation [20], the model performs better with low-resolution images. This is why another neural network model was implemented in the preprocessing phase to adapt the input images to an optimal resolution. This model automatically resizes custom images to resemble the training data as closely as possible, increasing the performance of images found in the wild. However, this does not guarantee success in all cases.

As discussed in this chapter, there are numerous architectures in the domain

of deep learning, each meticulously designed to fulfill a specific function. In the context of image recognition, CNNs represent a prominent class of algorithms that is extensively employed in contemporary OCR tools. A CNN is composed of a sequence of interconnected layers, whose structure and parameters play a crucial role in determining the model's performance during training. Furthermore, the efficacy of a convolutional model depends to a significant level on the dataset employed during the training process, as the model learns patterns from the samples comprised in it. The subsequent section will provide a detailed exposition on the generation of images that will be utilized for the CNN implemented in this thesis.

Chapter 3

Dataset Generation

The objective of this study is to propose a method capable of identifying machine-printed characters, with special attention to digitized or printed text. To achieve this, a selected CNN model will be trained on a comprehensive dataset comprising character images that exhibit a wide range of typographical variations, including differences in font style, size, aspect ratio, and image quality. In order to enhance the model's generalization capability across diverse printed-text representations.

A well-curated dataset can enable a relatively simple model to outperform a more complex one trained on poor-quality data. Recognition performance is influenced not only by the feature extraction and classification methods used but also by the quality and size of the training data. Factors such as dataset size, font styles, background variability, color, and contrast play a significant role in building a robust and generalizable dataset, and they will be taken into account during the dataset creation process.

This chapter focuses on the creation of a dataset composed of images containing alphabetic and numeric characters, mathematical symbols, and selected Greek letters, specifically tailored for applications in electronic circuit design.

3.1 Dataset Generation

The generation of a suitable set of images constitutes a significant challenge, given the requirement of neural models for large volumes of data, whether typed or handwritten, paired with accurate ground-truth labels for optimal performance. The manual production of these labels is a costly and time-consuming process.

To facilitate effective model training, it is essential that each generated image is accurately associated with a corresponding label denoting its class. To address this requirement, the process of image generation and label assignment was automated through the development and implementation of a dedicated algorithm in MATLAB. This automation is designed to ensure consistency, efficiency, and scalability in the creation of a labeled dataset suitable for training and validating of the recognition model. The algorithm is presented in the following section.

3.1.1 Character Generation Algorithm

The script automates the generation of images in PNG format, employing LaTeX rendering to depict characters or mathematical symbols. The character representation can be specified as standard text symbols or mathematical expressions. The fonts employed for this purpose are typically classified within a particular font-family, such as "Mathpazo", "Fourier", "Unicode-Math", "Euler", "Times New Roman", or "Palatino", thereby ensuring a greater diversity of samples.

The subsequent stage of the process is to define the list of characters to be created. The function will then iterate through each element until it generates a separate LaTeX file containing that symbol, and then compile it into a PDF using the specified LaTeX compiler. LaTeX provides three distinct approaches to document compilation: pdflatex, Xelatex, and lualatex. To satisfy the requirements of this feature, Xelatex was selected based on its ability to compile characters from the majority of the selected fonts. Finally, the resulting PDF format is converted to a PNG image.

This **Algorithm 1** is a particularly effective tool for creating a visual dictionary or a dataset of symbols rendered in LaTeX using different fonts. The characters that comprise the dataset are upper and lower case letters of the alphabet, numerals, fundamental mathematical symbols, and a selection of Greek letters, in both lower- and uppercase, that can be utilized as variables within certain electronic circuits.

Similarly, given that the majority of the font families utilized in **Algorithm 1** yielded analogous visually font styles for the Greek letters, it was imperative to explore alternative fonts capable of offering greater variability in character generation. Consequently, the generation of selected Greek letters was facilitated through the utilization of LaTeX Equation Editor available in Sciweavers [21], an open-source,

Algorithm 1 Generate Symbol Images from LaTeX

Input: `math_mode`, `font_name`, `compiler_type`
Output: PNG images of LaTeX-rendered symbols
Set `character_list` depending on `math_mode`
Initialize `math_dict` and `filename_dict` maps
Define output directory and ensure it exists
for all `symbol` **in** `character_list` **do**
    if `symbol` ∈ `math_dict` **then**
        `filename` ← `filename_dict[symbol]`
        `latex_symbol` ← `math_dict[symbol]`
    else
        `filename` ← infer name from `symbol`
        `latex_symbol` ← `symbol`
    end if
end for
if `math_mode` is `False` **then**
    Generate a standalone LaTeX file with `symbol`
else
    Generate LaTeX math file with `latex_symbol`
end if
Compile using selected `compiler_type`
Convert PDF to PNG
Clean auxiliary files

online tool that furnishes a compendium of fonts endowed with mathematical functionality, different sizes, colors, and supports several file formats. The total number of new images generated with this tool for each Greek letter was 14.

Pattern matching studies show that recognition performance is not only influenced by the choice of features and classification algorithms, but also by the quality and quantity of the training data.

To develop a robust character recognition dataset, it is essential to include a substantial number of character images that encompass a diverse range of font styles, deformations, and typographic variations, such as italic, bold, and regular forms. Ensuring that the model can effectively distinguish between these stylistic differences is critical for accurate and efficient character identification within images. However, manually generating approximately 100,000 images along with their labels is a time and resource-intensive process. To address this, the present work utilizes the **Chars74K-Fonts** dataset to streamline the image generation process.

3.1.2 Chars74K-Fonts Dataset

This open-source dataset [22] contains 62,992 character images categorized into 62 classes, representing alphanumeric characters (0-9, a-z, A-Z) of which 10,160 are digit samples and 26,416 samples each for uppercase and lowercase. Each image

has a resolution of 128x128 pixels and contains a character rendered in black using computer-generated fonts on a white background. The data set includes four distinct typographic variations that represent different combinations of regular, bold, and italic font styles. The dataset has 7705 natural character images, 3401 hand-drawn characters using PC, and 62,992 computer-synthesized fonts. **Figure 3.1**, show some character examples used in the dataset

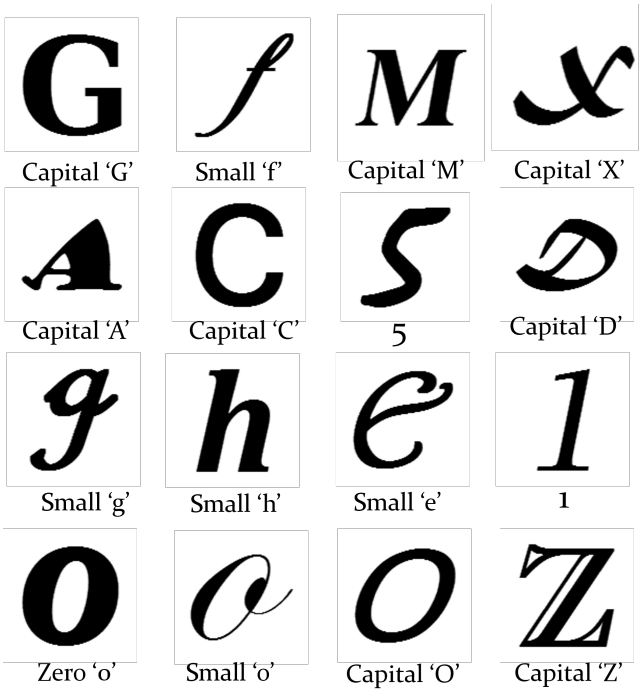


Figure 3.1: Samples from Char74K-Font Dataset.

3.1.3 Data Augmentation

Data augmentation is a technique that is both efficient and cost-effective for increasing the size of a dataset. Recent research [23, 24] has demonstrated that augmented data can replicate the essential characteristics of real-world data. This method is widely used to enhance the diversity of training data, thereby improving the model's ability to distinguish features across varying scenarios. As a result, it enables the acquisition of more robust feature representations with unseen data, which ultimately contributes to improved system performance on new tasks.

Therefore, to increase the variability in the image set, the following transformations were applied to the preexisting data set using the `Albumentations` library: geometric transformations, color space adjustment, noise injections to emulate real-world scenarios, pixelation, and other techniques. A duplicate image was generated for each input image along with its label, as shown in **Figure 3.2**. Consequently, the image and the duplicate will be fed into the neural network.

3.1.4 Geometric Transformations

- **Rotation**: Assists the model in recognizing characters that may appear slightly slanted due to camera misalignment, writing slant, or document skew. Improves rotational invariance and generalizes better to non-ideal input.

  To introduce a rotation effect during the image augmentation process, the predefined function `"ShiftScaleRotate"` within the `Albumentations` library was employed. The parameter `shift_limit` was set to 0.0625 (i.e., 6.25%) to control horizontal and vertical translations, representing a fraction of the images height and width. To simulate rotational variance, a maximum rotation angle of 45° was applied, resulting in a tilting effect that mimics changes in camera perspective or object orientation. The transformation was configured to be applied with a probability of 100% (p=1.0), while all other parameters were maintained at their default values. As a result, **Figure 3.2** illustrates the visual changes introduced by this augmentation.

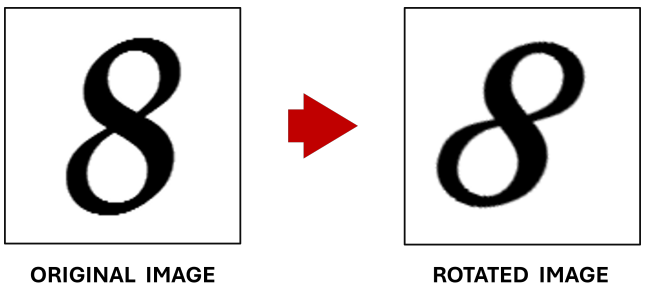


Figure 3.2: Rotation applied to the images.

- **Tilt**: Simulates oblique angles and perspective distortions, which are common in natural scene text, on walls, or signs. This improves the model's ability to interpret characters when viewed from slanted viewpoints. The tilt function was established with a factor of 0.2 (i.e. 20%) in order to guarantee a noticeable but subtle effect, achieved by shearing the image horizontally in proportion to its vertical position. As depicted in **Figure 3.3** , a comparison is drawn between the original image and the transformed image, highlighting the alterations made during the transformation process.

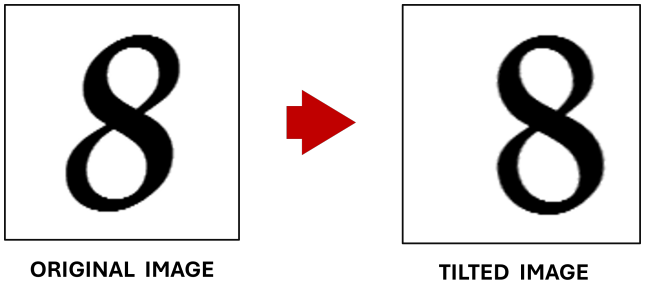


Figure 3.3: Tilted transformation applied to the images.

3.1.5 Color Space Adjustments

- **Brightness**: Improves the models robustness to varying lighting conditions by encouraging it to focus on shape rather than intensity, making it effective on underexposed and overexposed images. The `RandomBrightnessContrast` function was used in this context with its default parameters, introducing random variations in brightness and contrast within a range of ±20%. This means that some images may appear brighter or darker, while others may exhibit increased or reduced contrast. These variations help to create a more diverse and robust data set.

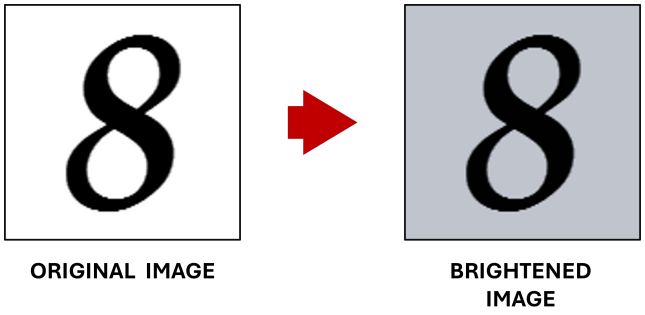


Figure 3.4: Brightness transformation applied to the images.

- **Hue-saturation**: Simulates color variations due to different backgrounds, lighting, or scanning artifacts. Allows the model to differentiate between characters that may be printed in different colors or placed on colored backgrounds. The default parameters were used for the `HueSaturationValue` function to create several images with different shades of colors, thus preventing overfitting to the exact hues and saturation of the training set. The resulting augmentation is shown in **Figure 3.5**.

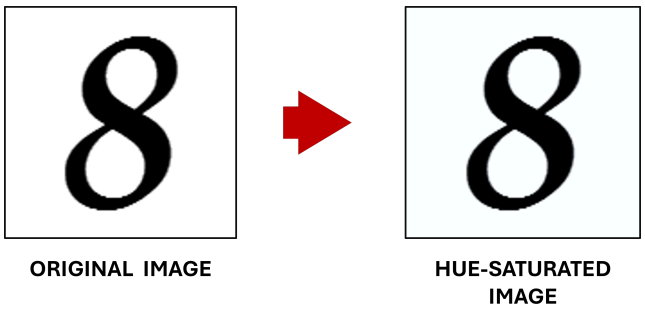


Figure 3.5: Hue-Saturation applied to the images.

3.1.6 Noise Injections

- **Gaussian Noise**: This process assists in the modeling of generalization by guiding it to prioritize essential structural patterns over individual pixel variations. This approach emulates real-world imperfections, including sensor noise, substandard images, and compression artifacts.

Gaussian noise is a statistical noise model that follows a normal distribution. It is frequently employed to simulate real sensor noise in image data. The normal distribution is defined by two parameters: the mean value of the noise (μ), which is 0, and the standard deviation (σ), which controls the dispersion of the noise values with respect to the mean. The standard deviation was set to 25, which resulted in a brightness fluctuation that was neither systematic nor regular, see augmentation is shown in **Figure 3.6**, but rather exhibited a more erratic and fluctuating pattern.

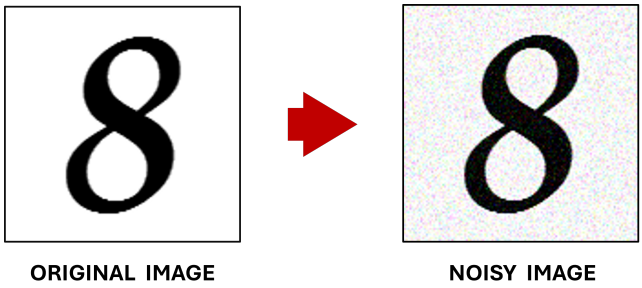


Figure 3.6: Noise transformation applied to the images.

3.1.7 Pixelation

- **Pixelation**: This technique degrades the resolution of the image by downscaling it and then scaling it back up to its original size, creating a "pixelated" effect. Increase the model's robustness to varying image quality. In this function, a pixelation factor of 10 was applied, yielding images characterized by moderately sized, blocklike artifacts that simulate reduced resolution, as shown in **Figure 3.7**.

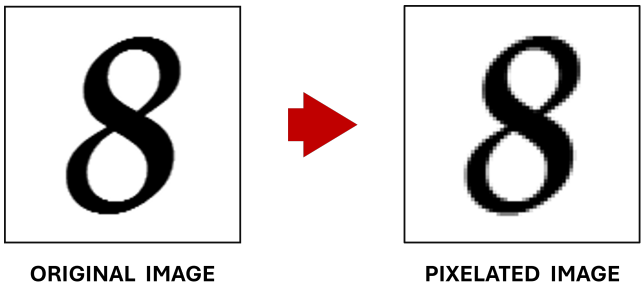


Figure 3.7: Brightness transformation applied to the images.

3.1.8 Other Techniques

- **Blurring**: Mimics motion blur or out-of-focus captures to encourage the model to learn the basic shapes of characters rather than relying on sharp edges or contours. The blur augmentation was implemented using the `Albumentations`

librarys `Blur` transform with a 7x7 kernel. The size of the kernel determines the extent of the neighborhood around each pixel that is used in the averaging process during image blurring. This choice of kernel size attenuates fine image details and produces a pronounced softening effect, as can be observed in **Figure 3.8**, thereby simulating a realistic sensor in the training data.

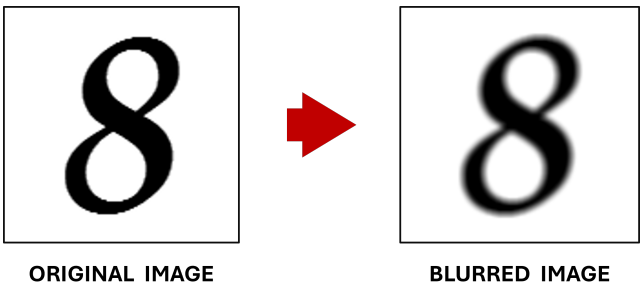


Figure 3.8: Blurring transformations applied to the images.

- **Distortion**: Applies perspective distortions to images to improve the robustness of the model to irregular deformations in character shapes. `ElasticDeformation` was applied using an initial global affine perturbation of magnitude 50, which introduced modest random translations, rotations, shears, or scaling of up to ś50 pixels or degrees, thereby creating coarse positional 'jitter'. Subsequently, a smooth displacement field was generated by sampling random x- and y-offsets and convolving them with a Gaussian filter with a standard deviation (σ) of 50. Due to the large σ, the resulting displacement field changes gradually across the image. Finally, the smoothed offsets were scaled by a factor of 1 (α), producing maximum per-pixel shifts of approximately 1 to 2 pixels over broad regions and creating gentle bulging distortions illustrated in **Figure 3.9**.

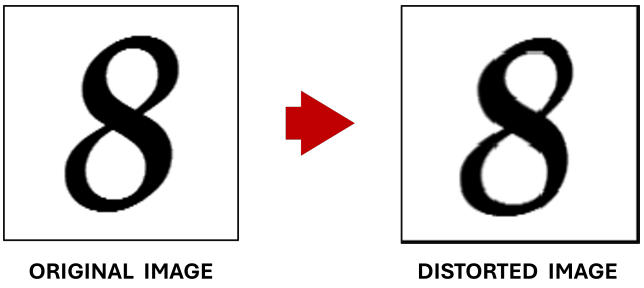


Figure 3.9: Distortion transformation applied to the images.

The final dataset comprises 94,381 images distributed across 258 classes, with resolutions ranging from 70x70 pixels to 128x128 pixels.

Chapter 4

Model Generation and Evaluation

Subsequent to the generation of the dataset, the next step is to define the Convolutional neural network model and the training process for the character recognition task. This chapter provides a detailed exposition of the preprocessing techniques that were applied to the images in the dataset. In addition, it describes the different tests that were performed using 'transfer learning' for the VGG16 model and a customized CNN model. These tests aimed to select the final model to be used during the inference phase. Finally, the Python libraries utilized will also be delineated in this chapter.

4.0.1 Packages

Before providing a detailed explanation of the functions incorporated within the script that constitute the final model and the training process, it is necessary to establish an overview of the libraries utilized in the implementation process.

TensorFlow was the principal software utilized in the development of this particular work. It is an open-source library focusing on the development of machine learning and artificial intelligence algorithms. TF was designed as a comprehensive framework, making it possible to construct convolutional neural networks (CNNs) at all levels, from the most basic to the use of pretrained models. Its main objective is to support the training and prediction of machine learning models and facilitate other data processing tasks.

Keras is a **TensorFlow** application programming interface (API) that simplifies and optimizes the development of neural network architectures and training processes. It provides a set of functions for the specification of hyperparameters, such as the learning rate, batch size, and number of epochs, as well as for the layer definition when developing the model workflow [25]. In addition, it supports both recurrent neural networks (RNN) and convolutional neural networks (CNN), making it suitable for a variety of deep learning applications.

Scikit-Learn is an open-source Python library offering a broad selection of tools for both supervised and unsupervised learning. Its core features include algorithms for classification, regression, and clustering, as well as techniques for dimensionality reduction. The library also provides various methods for model selection and performance evaluation, along with essential utilities for data preprocessing. The version employed was 3.10.

Imageio is a Python library that allows processing and interpreting images in different formats.

OpenCV2 is a freely available software library developed for computer vision and machine learning tasks, including image and video processing, object detection, and text recognition. The software offers a comprehensive range of functionalities that facilitate the manipulation of images and videos.

4.1 Script Description

4.1.1 Loading Dataset Function

The script begins with the function `load_dataset` described in **Algortihm 2**, generates two primary lists: **X** and **Y**, representing the images and their corresponding labels, respectively. Each element in the **Y** list indicates the class to which the associated image in **X** belongs. The **X** list contains a total of 94.381 samples, each of which undergoes a series of preprocessing techniques aimed at enhancing image quality. These techniques are applied to improve the overall efficiency and generalization ability of the machine learning model used in the subsequent stages. A detailed explanation of these preprocessing methods is provided in the following section.

Regarding the **Y** list, which contains the categorical class labels for each image, a **One-Hot encoding** approach is used to convert these categorical values into a numerical format. This step is crucial because most machine learning algorithms require numerical input for training and prediction and cannot directly process categorical data, such as text labels. **One-Hot encoding** is a standard solution to this limitation and ensures that the data set is compatible with a wide range of ML models.

Algorithm 2 Load Dataset and Pre-process Images

```
function Load_DataSet(path_dataset, target_size)
    Initialize empty lists X, y
    classes ← sorted list of folders in path_dataset, each folder represent a class
    class_to_index ← map each class name to a unique index
    for all class in classes do
        folder_path ← path to class folder
        for all image in class folder do
            img_path ← full path to image
            image ← read image as grayscale
            Preprocessed Image ← preprocessing_cnn(image, target_size)
            Append preprocessed image to X
            Append class_to_index[class] to y
        end for
    end for
    X ← concatenate X to stack arrays vertically in a list.
    y ← convert y to numpy array.
    y_onehot ← Apply One-hot Encode(y, depth=number of classes).
    return X, y_onehot, y, classes, class_to_index
end function
```

4.1.2 One-Hot Encoder

To facilitate a deeper understanding of the mechanisms underlying encoding, the following example is presented. The categorization of characters is achieved through

the utilization of a list comprising the respective names of each character, such as **'A', 'a_lowercase', 'B', 'b_lowercase', 'alpha', 'beta', 'sigma', 'minus', 'plus'**, and so forth. The classes mentioned above are encoded using the One-Hot method, whereby an explicit binary value is assigned to each category. As demonstrated in **Table 4.1**, this pattern guarantees that each categorical value has its own array of binary values 1 or 0, facilitating its integration into machine learning models.

Table 4.1: One-Hot Encoding of some characters used in the Dataset.

Character	**One-Hot Encoding**
A	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
B	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
C	[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
a_lowercase	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
b_lowercase	[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
c_lowercase	[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
α	[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
β	[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
λ	[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
minus	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
sum	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
point	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
other	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
...	...

4.2 Pre-processing Techniques applied to the Images

The application of specific pre-processing techniques is a key component to enhance the image quality. Given that the dataset under consideration is composed of typed-text characters, in which the majority of cases contain a white background, whilst other cases exhibit a contrasting, saturated color background or different sizes. The importance of standardizing the input images that will subsequently feed the CNN model cannot be underestimated, as it influences the models ability to consistently extract and learn discriminative features from each character, thus enhancing its generalization performance. The preprocessing function that has been implemented is presented in **Algorithm 3**.

The following list details the processing techniques applied to the images:

4.2.1 Resizing

The process of resizing images is a fundamental step in ensuring consistency and uniformity in the dimensions of all input data. This is a prerequisite for the effective functioning of machine learning algorithms. The OpenCV `resize()` method was implemented for the modification of image dimensions, by using interpolation

`INTER_AREA`, which is particularly effective in the context of downsizing images. The input size specified for the images was 64x64 pixels.

4.2.2 Normalization

In the context of the VGG16 model, the process of normalization is integrated within the Keras applications for the model (`keras.applications.vgg16`). The specific method is designated as `preprocess_input`, essentially involves the subtraction of the mean pixel value of each color channel BGR (Blue, Green, Red) derived during the training set of the ImageNet dataset, from the respective channels of each input image. Therefore, the resulting pixel distribution is centered around zero or within the range of 0 and 1. Facilitating convergence and consistency during the training process.

Algorithm 3 Preprocessing an image for CNN input

Require: Image, Target size
Ensure: preprocessed image is ready for CNN input
1: **if** Image has only two dimensions (grayscale) **then**
2: Convert image to 3-channel RGB by duplicating values
3: **end if**
4: Resize image to the target size using interpolation.
5: Apply VGG preprocessing to the resized image.
6: Convert pixel values to type `float32`
7: Expand dimensions so shape changes from $(H, W, 3)$ to $(1, H, W, 3)$.
8: **return** Preprocessed image

4.3 Pre-processing Techniques applied to the Bounding Boxes for Detection

To detect the bounding boxes successfully, it was also necessary to apply some pre-processing techniques. These techniques enhance the performance of the method `findcontours`, which is an OpenCV function used to identify the contours around an object in an image more effectively, thereby minimizing the influence of noise and artifacts that could compromise image resolution. The preprocessing steps implemented to improve bounding box detection are outlined in **Algorithm 4** and explained in the following sections.

4.3.1 Greyscale

Converting images to grayscale is a crucial pre-processing step that ensures uniformity in the color scale across the dataset. This standardization not only facilitates more consistent feature extraction but also reduces computational complexity by limiting the image data to a single channel. The `cvtColor` function from the OpenCV library was employed to perform the RGB-to-grayscale conversion.

4.3.2 Gaussian Blur

Blurring is a technique used to reduce unwanted noise in images by minimizing fine details. In this study, a Gaussian blur method was applied using a 5Œ5 convolution kernel present in the OpenCV library. This kernel size offers a balance between noise suppression and edge preservation. For instance, smaller kernels, such as 3x3, are capable of providing a lighter smoothing effect. Whilst the employment of larger kernels enhances clarity by reducing the blurring of the image. However, there is a risk that this will obscure fine details and merge closely positioned contours. Since there may be potential variations in lighting conditions in some images of the dataset, a moderately sized kernel (5x5) was selected, see **Figure 4.1**. It is effective for substantial noise reduction while retaining essential edge information.

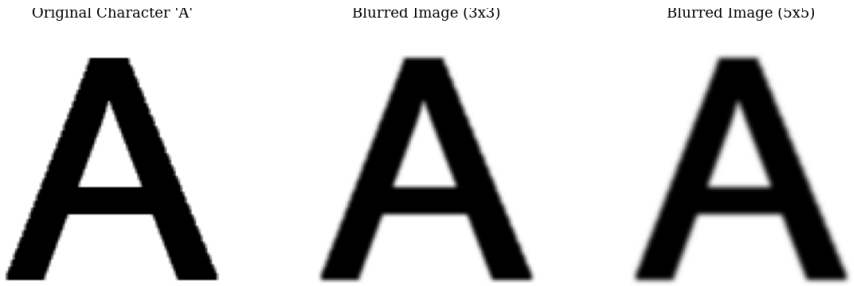


Figure 4.1: Gaussian Blurred Image.

4.3.3 Thresholding-Base Image

The subsequent technique applied to the images was binary thresholding with inversion. A threshold value of 156 was selected for the detection. The operation converts all pixel intensities falling below the threshold to black (i.e., the minimum pixel value of 0) and those falling above the threshold to white (i.e., the maximum pixel value of 255). This effectively inverts the image colour. This method established a boundary between the elements that make up the foreground and those comprising the background by rendering the former dark against the latter, which were rendered bright, as illustrated in **Figure 4.2**. This facilitates more accurate contour detection. The `threshold` method presented in OpenCV was implemented for this purpose.

4.3.4 Denoising Image

The next technique applied was non-local means denoising using the OpenCV function `fastNlMeansDenoising`. This method only works with grayscale images and retains the edge details and contours of each image after applying thresholding while filtering out residual noise, as shown in **Figure 4.3**.

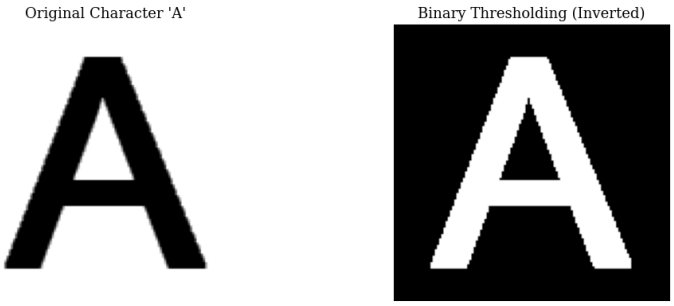


Figure 4.2: Thresholding-Base Image.

The method was configured with the following parameters: a `filter strength (h)` of 30. Medium to high filter values remove noise and small artifacts from previous processes; however, excessively high values ($h > 30$) may result in the removal of too many details, which is why a moderate value was chosen. Conversely, implementing a larger `template window` facilitates the identification of analogous pixel neighborhoods, thereby enhancing denoising quality. The value set in this field was 7. Concurrently, a reduced `search window` of 21 was selected for this task to improve computational efficiency without compromising denoising performance, as the noise pattern is relatively uniform across the image. For convention, it is recommended that the template window and the search windows should have odd numerical values.

These parameters ensure a robust balance between noise suppression and structural preservation, enabling a more reliable contour extraction in subsequent stages.

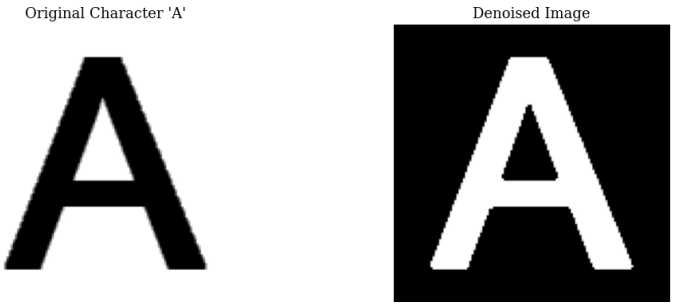


Figure 4.3: Denoised Image.

Algorithm 4 Preprocessing for bounding boxes detection

Require: Image, $kernel\_size(5, 5)$
Ensure: Denoised binary image is ready for prediction.
1: Convert the input image from RGB to grayscale.
2: Apply a Gaussian blur with the given kernel size to reduce noise.
3: Apply binary inverse thresholding with a threshold default value of 156.
Resulting in a binary image where black represents the background, and the white pixels the character.
4: Apply Non-Local Means Denoising to the binary image.
5: **return** Image preprocessed for BB detection

4.4 Model Development and Experimental Progression

To develop a robust model for character detection in typed images, several architectures and training strategies were explored. The following section delineates the experimental process, highlighting key decisions, and discusses the performance and limitations of each approach.

4.4.1 Initial Experiments

Subsequent to defining the dataset comprising all possible classes, the next step was to define the Convolutional Neural Networks utilized for character detection. Numerous models are available for image recognition, with deep-learning CNN demonstrating the highest level of performance among other classifiers for this character recognition task. In this work, a model for detecting text annotations in a circuit will be proposed.

As a first approach, a transfer learning technique was employed, which involves reusing a pre-trained model to address a new problem. This method leverages the knowledge acquired from a prior task with a large dataset to improve generalization in a different task with limited data. Instead of learning entirely from scratch, the model builds upon previously learned patterns and representations, allowing it to adapt more efficiently and effectively to the new context.

For these initial experiments, the dataset used contained only the main characters (the alphabet, the digits 09, some mathematical symbols, and Greek letters), equivalent to **79,452 samples** with 91 classes. Of these, 64% of the dataset (50,848 samples) was used for training; 16% (12,713 samples) was allocated for validation; and the remaining 20% (15,891 samples) was reserved for testing.

The architectural model and training configuration utilized are outlined in the following section.

4.4.2 VGG-16 model

The VGG16 model, a convolutional neural network (CNN) pre-trained on the ImageNet dataset, was used for character detection in this study. As outlined in **Section 2.5.1**, the model comprises 16 layers with a learnable set of parameters. In the initial experiments, the first 12 layers (for attempt 1) and 6 layers (for attempt 2) of the VGG16 model were kept fixed. These early layers primarily extract low-level, generic features such as edges, textures, and simple geometric patterns. Freezing them helps preserve foundational feature extraction capabilities that are typically transferable across different image datasets. Additionally, this approach reduces the number of trainable parameters, thereby accelerating the training process and mitigating the risk of overfitting.

Enabling only the deeper layers, particularly the fully connected layers, to remain trainable means that the model can focus on learning task-specific representations. To further reduce the risk of overfitting, a dropout regularization technique is applied after each **dense layer** with a rate of 0.25. The dropout randomly deactivates a proportion of neurons during training to encourage the network to develop more robust and generalized feature representations.

Finally, the output layer is implemented with a softmax activation function. This function converts the raw network output into a probability distribution for the potential classes, allowing the model to make probabilistic predictions for multi-class classification tasks.

The final architectural structure applied in this first approach is summarized as follows.

4.4.3 APROACH 1: VGG16 with 12 Frozen Layers

12 Pretrained Layers

Conv4_3: 512 filters, $3x3$ kernel, ReLU activation.
Max Pooling (Pool4): $2x2$ pool size, stride 2.
Conv5_1: 512 filters, $3x3$ kernel, ReLU activation.
Conv5_2: 512 filters, $3x3$ kernel, ReLU activation.
Conv5_3: 512 filters, $3x3$ kernel, ReLU activation.
Max Pooling (Pool5): $2x2$ pool size, stride 2.

Flatten

Dense (256): 256 units, ReLU activation. **Dropout (0.25)**
Dense (number_classes): number_classes units, Softmax activation.

4.4.4 APPROACH 2: VGG16 with 6 Frozen Layers

6 Pretrained Layers

Conv3_1: 256 filters, $3x3$ kernel, ReLU activation.
Conv3_2: 256 filters, $3x3$ kernel, ReLU activation.
Conv3_3: 256 filters, $3x3$ kernel, ReLU activation.
Max Pooling (Pool3): $2x2$ pool size, stride 2.

Conv4_1: 512 filters, $3x3$ kernel, ReLU activation.
Conv4_2: 512 filters, $3x3$ kernel, ReLU activation.
Conv4_3: 512 filters, $3x3$ kernel, ReLU activation.
Max Pooling (Pool4): $2x2$ pool size, stride 2.

Max Pooling: $2x2$ pool size, stride 2
Conv5_1: 256 filters, $3x3$ kernel, ReLU activation.
Conv5_2: 256 filters, $3x3$ kernel, ReLU activation.

Conv5_3: 91 filters, $3x3$ kernel, RELU activation
Flatten
Dense (256): 256 units, ReLU activation. **Dropout (0.25)**
Dense (256): 256 units, ReLU activation. **Dropout (0.25)**
Dense (number_classes): (Number of classes) units, Softmax activation.

4.4.5 Customized CNN

A tailored deep convolutional neural network architecture was created using Keras **Sequential** stacking, a method utilized by well-established models to add layers one after the other in a linear fashion, where the output of one is the input to the next. The initial block of convolutional layers was selected to extract fundamental visual features such as edges, lines, and textures, applying 128 filters. In the subsequent block, the number of filters was increased to 256, enabling the network to learn more complex representations, including corners and motifs, by integrating lower-level features. The final convolutional block was configured with 512 filters. It was designed to capture high-level abstract representations that may correspond to components, such as object or character structures. The structural architecture of this tailored CNN was inspired by the predefined VGG16 model, as illustrated in **Figure 4.4**.

Subsequent to the convolutional blocks, a flatten layer was incorporated to convert the 3D output of the max pooling layer into a 1D output. Two dense layers (fully connected layers) with sizes of 512 units and 1024 units were added, using the activation function 'RELU'. Likewise, to prevent overfitting, a dropout layer was incorporated between each fully connected layer as a regularization technique. The ultimate classification layer incorporates the 'Softmax' activation function, with the output size being equivalent to the number of classes in the data.

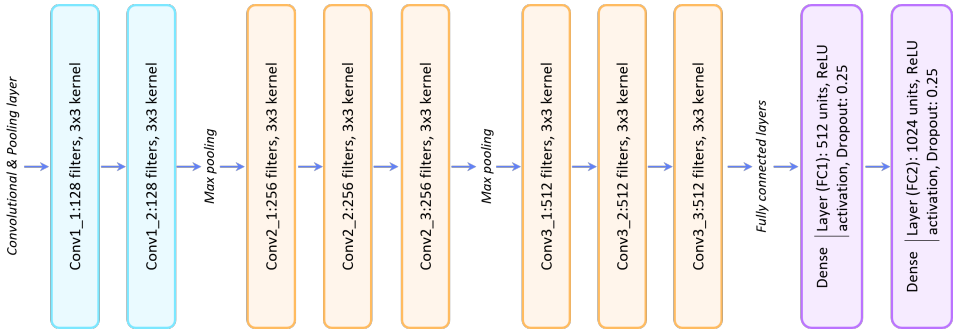


Figure 4.4: Customized CNN architecture.

Before being processed by the model, all input images were resized to a standardized resolution of $64x64$ pixels to ensure consistency in input dimensions.

The structural configuration of this model is presented in the following **Sec-**

tion 4.4.6.

4.4.6 Customized CNN Layers

Conv1_1: 128 filters, 3x3 kernel, ReLU activation, padding = "same"
Conv1_2: 128 filters, 3x3 kernel, ReLU activation, padding = "same"
Max Pooling: 2x2 pool size

Conv2_1: 256 filters, 3x3 kernel, ReLU activation, padding = "same"
Conv2_2: 256 filters, 3x3 kernel, ReLU activation, padding = "same"
Conv2_3: 256 filters, 3x3 kernel, ReLU activation, padding = "same"
Max Pooling: 2x2 pool size

Conv3_1: 512 filters, 3x3 kernel, ReLU activation, padding = "same"
Conv3_2: 512 filters, 3x3 kernel, ReLU activation, padding = "same"
Conv3_3: 512 filters, 3x3 kernel, ReLU activation, padding = "same"
Max Pooling: 2x2 pool size

Flatten
Dense Layer (FC1): 512 units, ReLU activation, **Dropout**: 0.25
Dense Layer (FC2): 1024 units, ReLU activation, **Dropout**: 0.25
Output Layer (output): (Number of classes) units, Softmax activation.

For the initial experiments, the three models were compiled using the Adam (Adaptive Moment Estimation) optimizer by default. This optimizer is renowned for its capacity to adapt the learning rate based on the average of the first and second moments of the gradients[26], often resulting in more rapid and stable convergence. Therefore, the learning rate was set to $1e^{-3}$ for the customized CNN and to $1e^{-5}$ for the VGG16 model with six and 12 frozen layers.

The loss function chosen was `Categorical Cross-Entropy`, which is widely used in multi-class classification tasks. The difference between the predicted probability distribution and the true one-hot encoded distribution is computed by this loss function. As each image in the dataset belongs to a unique class, the aim is to minimize the loss, ensuring the predicted probabilities closely align with the actual labels. Likewise, performance was evaluated using `Accuracy` metric, which measures the proportion of correctly classified samples in each epoch.

The training dataset comprised 50,848 samples (64%), which were utilized across three distinct models: a customized CNN, the VGG16 model with 6 frozen layers, and the VGG16 model with 12 frozen layers. The validation set consisted of 12,713 (16%) samples, while the test set included 15,891 samples (20%).

4.5 Training Function

The `train_cnn_model` function was designed to train a CNN using a given dataset of input images and their corresponding class labels. It has two purposes:

- First, it prepares the dataset by converting it into the correct numerical format and partitioning it into training, validation, and test subsets.
- Second, it initializes the CNN model with the correct input and output dimensions.

Initially, the input data corresponding to the images, denoted by `X`, and the labels (`y_onehot` and `y_index`) are converted into NumPy arrays with the correct data types, step 1 in **Algorithm 5**. The feature matrix `X` and one-hot encoded labels `y_onehot` are both cast to "$float32$" to guarantee compatibility with the categorical cross-entropy loss function. Likewise, to preserve the distribution of classes during the process of stratified sampling, the class indices (`y_index`) are converted to "$int32$".

In the subsequent stage of the procedure, the dataset is divided into three subsets by means of a two-stage stratified splitting procedure (step 2 in **Algorithm 5**). In the initial phase, 20% of the data is designated as a test set, thereby ensuring that the class distribution in the test subset matches that of the original dataset. The remaining 80% is stored as a temporary subset. In the subsequent phase, this provisional subset is subjected to further segmentation, with 64% of the total dataset allocated to the training set and 16% to the validation set. This process employs stratification to ensure the preservation of balanced class distributions. The resultant data is divided into three partitions: **64% for training, 16% for validation, and 20% for testing**. This configuration is a common practice in machine learning, as it facilitates both the optimization of models and the evaluation of performance in an unbiased way.

Following the division of the data, the model is initialized. Multiple monitoring and optimization strategies are configured to track the process, such as *EpochLogger*, which systematically saves the metrics (train and validation accuracy and loss) at an epoch level, step 4 in **Algorithm 5**.

Training is then executed using the 'fit' method (step 5 in **Algorithm 5**). In accordance with this method, certain hyperparameters were established, including the batch size, which was set to the typical value of 32, and the number of epochs, set at 50. In machine learning, batch size refers to the number of training samples that the model processes to calculate predictions. These predictions are then compared with the true labels, where the error (loss) is calculated. This process continues until all the samples in the training set have been used, marking the completion of one epoch.

Similarly, the **'shuffled'** option was enabled in the 'fit' method in each epoch to enhance generalization, and the validation set was employed to monitor the model's progress in each epoch. The training process was stopped after the validation accuracy metric did not improve after five epochs. The model is then saved for future use and reproducibility.

The implementation of this function was conducted for all model architectures proposed in **Sections 4.4.3, 4.4.4, 4.4.5**.

Algorithm 5 Train a CNN model function `train_cnn()`

Require: Feature data X, One-hot labels y_onehot, Class indices y_index, Mapping dictionary $class\_to\_index$, Optional parameters: $batch\_size$, $epochs$
Ensure: Trained CNN model saved to disk.
Step 1: Convert input data to correct types
  $X \leftarrow$ float32 NumPy array.
  $y\_onehot \leftarrow$ float32 NumPy array.
  $y\_index \leftarrow$ int32 NumPy array.
Step 2: Split data into Train, Validation, and Test sets
  Split X and y_onehot into 80% Training+Validation and 20% Test (stratified by y_index)
  Split Training+Validation into Training (64%) and Validation (16%) sets.
Step 3: Create CNN model
  cnn_model(input_shape, number of classes)
  Print model summary.
Step 4: Setup logging and callbacks
  Initialize training log file.
  Create custom callback for per-epoch logging.
Step 5: Train the model
  Call $model.fit()$ with:
    Training data (X_train, y_train)
    Validation data (X_val, y_val)
    Batch size, epochs, callbacks, verbose=2, shuffle=True.
Step 6: Save trained model
  Save model to $model\_save\_path$

4.6 Results of First Models Experiments

The comparative results of the three models, Customized CNN, 12 Layer Frozen, and 6 Layer Frozen, are illustrated in **Table 4.2**, demonstrating notable differences in performance across the training, validation, and test phases.

The 6 Layer Frozen model exhibited superior performance in terms of overall generalization, attaining the lowest test loss (0.0814) and the highest test accuracy (0.9732), while exhibiting the fewest total errors (426) and unique errors (62). In contrast, the 12-layer frozen model demonstrates the poorest performance, exhibiting

Table 4.2: Comparison results of CNN, 12 Layer Frozen, and 6 Layer Frozen Models during Testing phase.

Metric	Customized CNN	12 Layer Frozen	6 Layer Frozen
Train Loss	0.0341	0.0748	0.0288
Train Accuracy	0.9864	0.9718	0.9879
Validation Loss	0.0842	0.1132	0.0668
Validation Accuracy	0.9707	0.9633	0.9781
Test Loss	0.0929	0.1175	0.0814
Test Accuracy	0.9682	0.9600	0.9732
Total Errors	506	635	426
Unique Errors	104	122	62

the highest test loss (0.1175), the lowest test accuracy (0.9600), and the greatest number of errors and unique errors. Although the customized CNN model achieves robust results, with a test accuracy of 0.9682 and moderate error counts (506 in total, 104 of which are unique), it is marginally outperformed by the 6-layer frozen configuration. This suggests that partial fine-tuning (freezing only six layers) provides a favorable balance between transfer learning and model adaptability.

Table 4.2 also reports the number of unique classification errors made by the model, referring to distinct types of misclassifications observed during the testing phase. The total error count reflects the frequency with which each misclassifications occurred. For example, if the character $'l'$ or $'O'$ was incorrectly classified as $'1'$ or $'0'$ on multiple occasions, each instance contributes to the total error count, the total errors out of 15.891 test images. In contrast, the *uniqueerror* count captures only the presence of a particular type of mis-classification (i.e., how frequent a class was incorrectly classified), regardless of the number of times it occurred.

Since the results achieved by the proposed CNN were robust and comparable to those obtained with the pre-trained VGG16 model using six frozen layers, this suggests that the application of fine-tuning techniques, such as early stopping and learning rate annealing, may further improve generalization in custom models and potentially yield superior performance. Accordingly, the following section describes the fine-tuning strategies applied to the customized CNN.

4.7 Model Fine-Tuning the Customized CNN

To refine the proposed Convolutional Neural Network (CNN) model and mitigate excessive overfitting during training, several fine-tuning techniques were implemented in the training method described in the **Section 4.5**. These strategies also contributed to improving the model's overall accuracy. The applied methods were employing Keras callbacks, and they are presented below:

1. **Learning Rate Adjustment:** This parameter controls the step size at which the model recalibrates its weights during the training phase. The correct selection of this parameter is therefore essential to avoid unstable training or the convergence to sub-optimal solutions. In the initial experiment, a learning rate of 1×10^{-3} was employed during the training stage. This value could be high for a customized neural network, since it exhibited early overfitting in comparison with the VGG16 model. In order to address this issue during the fine-tuning process of the model, a lower learning rate of 10 was adopted, with the objective of achieving a more gradual and stable convergence.

2. **Early Stopping:** Early stopping is a regularization technique that is used to prevent overfitting by terminating the training process once the model's performance on a validation set has stopped improving. Typically the selected metric corresponds to validation loss. If there is no improvement over a specified number of consecutive epochs (patience parameter) training process ends, and the model weights corresponding to the lowest validation loss are automatically recovered. In this case, for the customized CNN a patience value of five epochs was implemented.

3. **ReduceLROnPlateau:** This method serves as a model optimization by reducing the learning rate when performance improvements stall, thus minimizing the risk of overfitting. By applying a controlled reduction, the model performs more refined updates of the weights, improving its convergence as the training progresses. Particularly, the learning rate was adjusted based on the validation loss metric. If this metric stops improving for 3 consecutive epochs, the learning rate is halved, with a predefined minimum threshold of 1×10^{-6}.

4. **Model Check Point:** The ModelCheckpoint method is used to preserve the most optimal version of the model achieved throughout the training process. The selection is determined based on a specified evaluation metric, such as validation loss or validation accuracy. The model whose weights demonstrate the most optimal performance according to the selected metric is then saved. Adopting this technique will increase the likelihood of achieving superior predictive accuracy during the testing phase.

5. **Increasing dataset:** A new set of images was incorporated into the existing

dataset for this new experiment with the fine-tune model. The total number of samples used was 94,381. Images generated comprised subscripts and superscripts, particularly in instances involving resistors. capacitors, inductors, and voltages ($R_1, C_2, L5, etc$). Images found on the internet were also incorporated for a more robust dataset.

4.7.1 Results obtained after Model Fine-Tuning

This section presents the results obtained from applying the aforementioned model refinement techniques.

Table 4.3: Performance comparison between the VGG16 model with 6 frozen layers (baseline) and the fine-tuned customized CNN.

Metric	VGG16 (6 Frozen Layers)	Customized CNN
Train Loss	0.0114	0.0118
Train Accuracy	0.9949	0.9954
Validation Loss	0.0387	0.0424
Validation Accuracy	0.9877	0.9880
Test Loss	0.0399	0.0404
Test Accuracy	0.9887	0.9886
Total Errors	258	256
Unique Errors	70	65

As demonstrated in **Table 4.3**, the implementation of fine-tuning techniques on the previous model led to substantial improvements in accuracy across all training, validation, and testing scenarios. This was accompanied by a significant decrease in loss at each stage. These observations suggest improvements in training, generalization, and robustness. Furthermore, the customized CNN fine-tuned resulted in a significant reduction in classification errors, both in total and in uniquely misclassified samples, in comparison to the one observed in the prior experiment (**Section 4.3**).

The findings indicate that the architecture and training strategy employed in the CNN customized fine-tuning process were more effective in capturing the underlying structure of the data, including variations in font and style. Consequently, this approach yields predictions that are more reliable and generalizable than those obtained from experiments performed as a starting point.

4.7.2 Comparison with the baseline model

Furthermore, the results obtained with the pre-trained model and the customized CNN fine-tuned under the same conditions outperformed VGG16 with 6 frozen layers across most evaluation metrics; the performance margin between the two is relatively small. Since VGG16 is used as the baseline model to evaluate the performance of the proposed model, the results highlight a positive generalization for the fine-tuned CNN.

Similarly, as shown in **Table 4.3**, a slight discrepancy was observed in the validation test, with results of 98.8% and 98.7% for the customized CNN and VGG16 models, respectively. Both models demonstrated high effectiveness when evaluated on unseen data, indicating their ability to predict with new information. Likewise, it achieved a test accuracy of 98.86% (customized CNN) compared to 98.87% for the pre-trained VGG16.

These differences, although measurable, are minor when the advantages of custom architecture are taken into account. Having greater control over parameter settings and applying a lightweight model can be beneficial to the task. In practical terms, these discrepancies are within the range of run-to-run variability for models of this size, especially when considering stochastic factors (initialization, batch sorting, data augmentation, etc.). Similarly, the test loss of the **customized CNN** showed small discrepancies, 0.0404 vs. 0.0399 corresponding to the VGG16 model, and the error count differed slightly, with only 2 more total errors and 5 more unique errors (258 vs. 256). Therefore, they are within the expected variability of deeplearning runs.

A key benefit of **customized CNN** is its architectural flexibility and reduced computational complexity in contrast to VGG16, which is a deep and resource-intensive model with more than 138 million parameters (even after six frozen layers, the number of pretrained parameters would be $\gtrsim$60 M). The customized CNN was designed with a more compact structure, tailored specifically to the characteristics of the dataset, such as character recognition in varying fonts and styles, and has 25 million parameters. This enhancement in efficiency is particularly pronounced in environments characterized by limited computational resources.

This finding indicates that the fine-tuning techniques were sufficiently effective to optimize a lightweight model to a point that its performance is almost equivalent to that of a much deeper and pre-trained architecture. This was therefore the final model selected for making predictions about the characters in the subsequent stages. In addition, the optimized model was saved in the `*.h5` format to preserve its weights and parameters learned during the training phase, facilitating straightforward loading and deployment in other instances.

4.7.3 Weakness of the customized CNN

Certain errors were identified during the testing phase. These errors are primarily attributable to visually similar characters with which the model occasionally experiences difficulty in distinguishing.

The **Snippet 4.1** provides an illustration of the 10 classes in which the model generated the most classification errors, along with the frequency with which these

errors occurred during the testing phase.

Listing 4.1: Most Frequent Recognition Errors.

```
# Predicted Label    vs  True Labels     (Unique Errors)
-----------------------------------------------------------
1. Predicted: 0_zero      True: O          | Count: 84
2. Predicted: O           True: 0_zero     | Count: 44
3. Predicted: I           True: l_lower    | Count: 33
4. Predicted: l_lower     True: 1          | Count: 14
5. Predicted: l_lower     True: I          | Count: 13
6. Predicted: 1           True: l_lower    | Count: 10
7. Predicted: U           True: u_lower    | Count: 5
8. Predicted: k_lower     True: K          | Count: 5
9. Predicted: Y           True: y_lower    | Count: 4
10. Predicted: p_lower    True: P          | Count: 4
-----------------------------------------------------------
```

Chapter 5

Main Function

The main function is responsible for synthesizing all the steps necessary to implement the prediction of the annotations. In essence, it is the function employed for the reproducibility of the algorithm developed in this study.

The following steps constitute the primary function:

1. Load the CNN model.
2. Load the class labels list.
3. Execute the prediction phase for the character detection.
4. Execute the post-processing function to assign the meaning and label form to the characters predicted.
5. Convert the final label string into LaTeX format.
6. Log results obtained.
7. Return the final result.

5.1 Calling the main functions

The main function designated as `cnn_detection()` receives as input the image to be recognized and essentially executes the character detection and post-processing functions. The resulting string is then transferred to a method that converts the post-processed predictions into LaTeX format. These results are saved in a text file for subsequent analysis.

The returned output of this function is the formatted string representing the label of the circuit. **Algorithm 6** provides a comprehensive overview of the procedure implemented in this function. All these steps will be explained in detail in the following sections.

Algorithm 6 Main function call **cnn_detection()**

Require: Image path *img_path*
Ensure: Predicted text in LaTeX format
1: **STEP 1: Load CNN model**
Load the trained CNN model from the specified path.
2: **STEP 2: Load class labels**
Read class names from the class labels file into a list.
3: **STEP 3: Prediction Phase (predict_characters())**
Pass the input image to the CNN model to obtain the character predictions, bounding boxes coordinates and confidence scores.
4: **STEP 4: Post-process function (post_processing())**
Refine the predicted text via post-processing.
5: **STEP 5: Convert to LaTeX**
Transform the refined predictions into LaTeX-formatted text.
6: **STEP 6: Log results**
Save prediction logs including confidence scores and processing time.
7: **STEP 7: Return final output**
Return the final LaTeX-formatted text.

5.2 STEP 1: Load CNN model

In this step, the Convolutional model that will be used for prediction is loaded. In this instance, the model to be employed is the one defined before in **Section 4**, corresponding to the customized CNN.

5.3 STEP 2: Load class labels

Consequently, in this step, the labels of the classes employed in the dataset were loaded. In order to ensure reproducibility, it is recommended that the class labels be saved in a *.txt file. This will allow them to be loaded on any other computer without the need for the original dataset.

5.4 STEP 3: Prediction Phase (`predict_characters()`)

5.4.1 Algorithm

This function aims to execute the inference (prediction) phase using the annotations received by the circuit. It receives as input the following parameters:

- Image path,
- A list with the labels of the classes that have been defined in the model.
- The CNN model.

The output of this function corresponds to three *txt file that contain: the model-predicted characters, the prediction time, and the confidence level of the inference. Examples of these file are presented in **Snippet 6.19, Snippet 6.20, Snippet 5.4**.

STEP 1: Loading Image

The method is initiated by loading the input image. An example of a possible input annotation image is shown in **Figure 5.1**.

Figure 5.1: Input annotation image example.

STEP 2: Pre-processing image for contour detection

Subsequently, the image is passed to the preprocessing function for bounding boxes, to prepare it for the contour detection method. The latter must match the pre-processing steps employed during model training. This is due to fact that the model has been trained on pre-processed image data. Discrepancies between training data and inference preprocessing have the potential to compromise the integrity of the pipeline, resulting in a distribution shift, therefore, affecting the final predictions. The preprocessing steps applied are already described in **Section 4.3**. **Figure 5.2** depicts an example of an image after preprocessing steps.

Figure 5.2: Preprocessed image example.

STEP 3: Detecting Contours and sort contours from left to right.

Following this, contour detection is applied to the preprocessed image. This technique is employed to identify the candidate character region within the image by delineating a contour around it, as illustrated in **Figure 5.3**. The method was implemented using the openCV function `cv2.Findcontours()`, which is designed to detect all points having the same color and intensity, and return them as a contour, the threshold value was set to 156. These, in turn, are the representations of the shapes present in the image.

Figure 5.3: Output of cv2.Findcontours method.

Subsequently, the contours are sorted from left-to-right and top-to-bottom in order to maintain consistent spatial order. This preserve natural reading sequence in the image.

STEP 4: Bounding boxing (BB) generation.

On the other hand, the creation of the bounding boxes involved the implementation of a *for* loop, which was apply to systematically iterate through the list of contours that had been previously identified. The **cv2.boundingRect()** method, available in the OpenCV library, was applied in order to generate the rectangle around the shape of the identified object, see **Figure 5.4**. The function returns a lists of coordinates of the detected bounding boxes in the following format: The coordinates are specified as `[x_min, y_min, x_max, y_max]`. Where `[x_min, y_min]` represent the top-left-corner of a box and `[x_max, y_max]` the bottom-right-corner.

Figure 5.4: Bounding boxes generated.

STEP 5: Character Prediction

The final stage involves predicting the character contained within each bounding box. Another *for* loop was apply to iterate through the list of bounding boxes

detected in the previous step. To ensure that the extracted region adequately encompasses the character, a padding margin is applied around each bounding box. This was performed by modifying the BB coordinates accordingly.

The character was then subjected to cropping from the image using the BB coordinates. The extracted segments were passed to the preprocessing function for the input images to prepare it for the prediction phase. Once again, the function utilized for this scope, must be the same used during the training of the model. This was defined in **Section 4.2**.

Following preprocessing, the processed image segments are passed through the CNN model loaded before, using the `predict()` method for inference. The output of this stage are then saved in a *.txt file for analysis to be performed later. **Figure 5.5** shows a representation of the original image with the predicted characters.

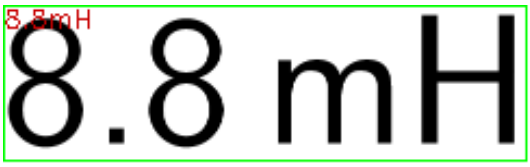


Figure 5.5: Final prediction.

STEP 6: Save results

The results comprise the predicted characters, together with the coordinates of the bounding boxes and the corresponding confidence levels. **Snippet 5.1** show the raw predictions obtained after character detection.

Snippet 5.1: Raw predictions output.

```
[(('8', 0.28), (2, 0, 13, 40)),
(('.', 1.0), (15, 19, 13, 26)),
(('8', 1.0), (32, 19, 13, 26)),
(('m', 1.0), (62, 1, 22, 36)),
(('H', 1.0), (98, 10, 23, 27))]
```

Algorithm 7 also provided a comprehensive overview of the steps involved in the prediction phase.

Algorithm 7 Character Prediction from Image (`predict_characters()`)

STEP 1: Load image
STEP 2: Pre-process image for contour detection
STEP 3: Detect contours and sort bounding boxes from left to right
STEP 4: Generate bounding boxes
STEP 5: Prediction of Characters

for each **box** in **bounding_boxes do**
    Apply padding and crop character region
    Pre-process the cropped image for CNN input
    Predict character using the trained model
    Extract predicted label string and confidence
    Store prediction with bounding box coordinates
return List of predicted characters, prediction time, and confidence scores

5.5 STEP 4: Post-process function post-processing()

In **Chapter 6**, comprehensive details on the post-processing function will be provided. This method essentially retrieves the predictions generated by the **predict_characters()** function and processes them in a way that ensures the resulting output aligns with the specified annotation label format. The function's output is a string containing the annotation.

5.6 STEP 5: Convert to LaTeX format

The outcomes of the post-processing method occasionally yield a sequence of characters that does not align with the LaTeX cleaner format. In this research project, the notations recognized by the CNN + post-processing function and Pix2Tex methods must be written in a format compatible with LaTeX format, as these predictions will later be used to compare the two methods.

For this reason, the `convert_to_latex()` function was developed. This function receives the post-processed predicted tokens (character string) as an input parameter and converts them to LaTeX format. It identifies numerical expressions, fractional values, Greek letters, electrical units, and scientific notation, and reformat them in order to produce correct cleaner LaTeX syntax. The 're' library in Python was implemented to generate the regular expression required to identify and extract specific structures within a text string.

For instance, when processing a label such as `"alpha_{1}i_{22}"`, a corresponding regular expression is then constructed to identify this specific pattern of characters. If a match is found, the function automatically replaces the Greek word with its LaTeX equivalent, denoted by `"\alpha"`, thus ensuring that the label is correctly

formatted as "`\alpha_{1}i_{22}`" in the final output. The same logic was applied to the rest of the cases mentioned before.

In this transformation, mathematical commands and markups associated with the LaTeX format are not included. The final result must exclusively maintain the structure of LaTeX expression, incorporating elements such as subscripts, superscripts, and fractional formats.

Finally, the resulting data is stored in a text file. **Snippet 5.2** depicts the resulting output after applying the conversion.

Snippet 5.2: Example of reformatting performed.

```
Input                 Output
67 Omega          ->  67 \Omega
alpha_{1}i_{22}   ->  \alpha_{1}i_{22}
phi_{3}i_{5}      ->  \Phi_{3}i_{5}
3/4 V_{2}         ->  \frac{3}{4}V_{2}
90.10^{-3}i_{6}   ->  90\cdot10^{-3}i_{6}
```

An example of the resulting final predictions, following post-processing and conversion to a LaTeX-like syntax, is presented in **Snippet 5.3**.

Snippet 5.3: Post-processed predictions after LaTex conversion.

```
(5)
\frac{3}{2}A
8.8mH
(1)
(3)
\frac{6}{17}A
\frac{20}{17}H
(0)
(6)
\frac{5}{18}V
\frac{5}{4}\Omega
13\Omega
V_{9}
i_{9}
6.3V
41mH
37\Omega
7.3mH
```

5.7 STEP 6: Log results

After converting the predicted label into the standard LaTeX format, the information is saved in a text file for control information purposes. The content of this

file corresponds to the following elements and they are depicted in **Snippet 5.4**:

- the list of predicted characters resulting from the **prediction_characters()** function;
- the predicted label after post-processing; and
- the time taken for the entire prediction process, which is the sum of the prediction time and the post-processing time.

Snippet 5.4: List of predicted characters

```
Initial_prediction:[(('(', 1.0), (2, 4, 21, 85)), (('5',
    1.0), (32, 11, 34, 58)), ((')', 1.0), (75, 4, 21, 85))]
After postprocessing:(5)
Time after pp: 0.31528615951538086 seconds

Initial_prediction:[(('-', 0.83), (3, 46, 33, 6)), (('2',
    1.0), (6, 61, 26, 40)), (('3', 1.0), (7, 0, 25, 40)), ((
    'A', 1.0), (59, 11, 50, 61))]
After postprocessing:\frac{3}{2}A
Time after pp: 0.46279072761535645 seconds

Initial_prediction:[(('(', 1.0), (2, 4, 21, 86)), (('1',
    1.0), (35, 10, 29, 59)), ((')', 1.0), (75, 4, 20, 86))]
After postprocessing:(1)
Time after pp: 0.3351309299468994 seconds

Initial_prediction:[(('(', 1.0), (3, 4, 21, 85)), (('3',
    1.0), (32, 10, 35, 60)), ((')', 1.0), (76, 4, 21, 85))]
After postprocessing:(3)
Time after pp: 0.35938501358032227 seconds

Initial_prediction:[(('-', 0.82), (4, 50, 61, 5)), (('1',
    1.0), (8, 65, 23, 39)), (('6', 1.0), (21, 2, 25, 40)),
    (('7', 1.0), (36, 65, 27, 39)), (('A', 1.0), (88, 15,
    51, 59))]
After postprocessing:\frac{6}{17}A
Time after pp: 0.5673356056213379 seconds

Initial_prediction:[(('6', 1.0), (1, 2, 35, 60)), (('omega'
    , 1.0), (56, 0, 55, 62))]
After postprocessing:6\Omega
Time after pp: 0.24060463905334473 seconds

Initial_prediction:[(('5', 1.0), (0, 6, 34, 59)), (('.',
    1.0), (46, 55, 7, 8)), (('8', 1.0), (65, 5, 34, 60)), ((
    'V', 1.0), (119, 3, 51, 60))]
After postprocessing:5.8V
```

```
Time after pp: 0.0001556873321533203 seconds

Initial_prediction:[(('5', 1.0), (0, 8, 35, 58)), (('.',
    1.0), (47, 56, 7, 8)), (('4', 1.0), (63, 7, 39, 57)), ((
    'm', 0.93), (124, 25, 53, 39)), (('A', 1.0), (186, 4,
    50, 61))]
After postprocessing:5.4mA
Time after pp: 0.00020241737365722656 seconds
```

5.8 STEP 7: Return final output

The function **cnn_detection()** returns the final predicted label string of the annotated image. For instance, in the case of the example illustrated in STEP 3, the final output string would be `8.8mH`.

This function is of vital importance when considering applying this application to other projects. In essence, it is the function that is utilized to make predictions using the CNN model trained in this study and the developed post-processing function.

Chapter 6

Post-Processing Function

This chapter delineates the functions associated with post-processing applied to the characters predicted by the Convolutional model. The CNN model generates a list of raw character predictions extracted directly from the annotation image received. These predictions must be verified and adjusted if any errors occurred during the recognition phase.

Post-processing involves the refinement of preliminary model results through the adjustment of specific pre-processing parameters, with the objective of enhancing prediction accuracy. The CNN model is employed to formulate a new prediction, with the parameters being adjusted accordingly. As a result, the new prediction generated may be the same as the previous one or show a character that had not been correctly identified during the first prediction. In the final stage of the process, the function assigns the context to the label according to the characters that were previously predicted.

6.1 Post-Processing Function

6.1.1 Overview

The `post_processing()` function has been developed to refine the raw predictions generated by the CNN model for circuit-related labels (such as voltages, currents, resistances, and component identifiers). The **predict_characters()** (**Section 5.4**) method generates an initial list of predicted characters that often exhibit misclassifications arising during the prediction process, since the model is susceptible to misclassifications for visually similar characters.

This function applies a set of rule-based conditions to correct these issues and produce context-format labels. On the other hand, the function also stores the processed results in a *.txt file. In order to obtain a comprehensive overview of the post-processing function, the **Algorithm 8** delineates the primary steps involved in the detection and creation of each circuit label.

The summary of these steps is illustrated in **Algorithm 8**.

Algorithm 8 Post-Processing of CNN Predictions (`post_processing()`)

STEP 1: Load the input image from *img_path*

Extract predicted characters from the initial predicted list (first character, last character, and if available, second and third)

Initialize $merged\_text \leftarrow \emptyset$

STEP 2: Evaluate predictions against predefined conditions $C_1, C_2, \ldots, C_{21}$

for $i = 1$ **to** C_n:

if predictions satisfy condition C_i:

Apply correction rules for C_i

Optionally re-run character recognition on bounding boxes

Merge corrected characters into *merged_text*

If no condition is satisfied: re-predict characters directly from bounding boxes and merge into *merged_text*

STEP 3: Post-process the image using `pp_prediction_cases()`

Save annotated image

STEP 4: Consolidate the Bounding boxes into a single one, after postprocessing of the characters predicted initially.

STEP 5: Log refined label into output ***.txt** file.

Log confidence scores into confidence file.

Compute post-processing time *prediction_pp_time* .

return (*merged_text*, *prediction_pp_time*)

6.2 Processing Steps

The subsequent section will provide a comprehensive explanation of the function.

6.3 STEP 1: Inputs of the Post-Processing method

- **predictions:** The initial list of characters predicted by CNN in the prediction method. Each element is a tuple that contains the predicted character and its corresponding bounding box.
- **img_path:** Path to the image labels.
- **model:** The CNN model used for character recognition.
- **classes:** The set of possible output classes (characters) the model can predict.

6.4 STEP 2: Logical Conditions Applied to Predictions

In this step, all the conditions were defined. Each condition was designed to determine a specific labeling pattern based on the list of predicted characters. Essentially, a set of conditions are evaluated to establish whether the characters on the list satisfy the patterns outlined in these conditions. Examples of these patterns include dependent sources written with subscripts, scientific notation or superscripts, values with decimal numbers, symbolic values, circuit units, and misclassified symbols.

Table 6.1: Electronic Circuit Symbols with Meaning, Units, and Examples

Symbol	Meaning	Unit	Notes	Example
V	Voltage source	V	Independent source	$V_4 = 12$ V
v	Voltage source	V	Dependent source	$v = \beta_3 v_4$
R	Resistor	Ω	Passive element	$R = 1$ kΩ
C	Capacitor	F	Passive element	$C = 10$ μF
L	Inductor	H	Passive element	$L = 5$ mH
A	Current	A	Independent current source	$I_4 = 2$ A
i	Current	A	Dependent source	$i = 3I_4$
P	Power	W	Power delivered/consumed	$P = VI$
S	Switch	—	Switching element	-
E	Voltage source	V	Independent source	$E = 5$ V
F	Faraday	F	Capacitor unit	-
H	Inductance	H	Inductor unit	-
Ω	Resistance	Ω	Resistor unit	-
G	Conductance	S	Reciprocal of resistance	$G = 1/R$

In order to facilitate a better understanding of the subject, the conditions were created with the understanding that the numerical values, alphabetical letters, and Greek letters found in the annotations or labels of an electrical circuit possess a

meaning depending on the manner in which they are presented. **Table 6.1** provides a comprehensive overview of the respective meanings of each letter within the context of an electronic circuit.

Explanation of `make_new_prediction()` Function

Prior to the description of the conditions, it is necessary to explain the method that will be used in the following steps. This function performs the same actions as the function detailed in **Section 5.4** (`predict_characters()`). The sole discrepancy between the two lies in the input parameters that the former is capable of receiving.

The purpose of this function is to fine-tuning the preprocessing of BBs phase, to facilitate the simplification of the inference method, thereby allowing certain parameters to be modified simply by passing them as hyper-parameters when the function is invoked.

The new input parameters aggregated are the following:

- `order`: Case variable refers to two possible ways to sort the BB when using **cv2.findContour()** method during the inference. So that it facilitates read the sequence during the post-processing.
  - For order=0: The BB will be sorted from left-to-right and top-to-bottom.
  - For order=1: The BB will be sorted from top-right to button-left.

- `var`: The purpose of this variable is to finetune the parameters of the preprocessing of the bounding boxes stage, with a view to facilitating the character detection. The parameter is associated with the image binary **cv2.threshold()** technique applied during the preprocessing of BB steps (see section 4.3.3). The possible values for thresholding parameters are:
  - var=1 →keep the threshold in 158 (default, using during first prediction in predicted_characters()).
  - var=2 →reduce the threshold to 130, for a smooth definition in the characters, it makes the edges less stark.
- `pad`: Allow the modification of the quantity of padding added during the generation of BB.

The modification of certain parameters in order to detect the Bounding Boxes technique has been shown to facilitate the identification of the contours surrounding the characters, thus ensuring the generation of the Bounding Boxes in an optimal manner. It is imperative to emphasize that this step was incorporated for the purpose of fine-tuning the post-processing. Improves bounding box quality; doesnt alter

CNNs learned features.

6.4.1 CONDITIONS

The conditions were structured into a series of rules based on input predictions. The following segment will provide an explanation of the most relevant conditions established in the postprocessing function to illustrate the logic behind this rules, with the presentation of a corresponding example.

CONDITION 1: Component annotations including numerical subscripts.

Condition 1 has been specifically designed to detect annotations containing numeric prefixes only for voltages, currents, capacitors, resistors, or inductors. This notation is typically used in dependent sources. The label considering in this condition includes only integer numbers in the numeric part of the label, such as: $3V_7$, $3R_5$, $112i_{13}$, $23C_2$.

The following conditions must be met for the instruction to be executed:

- **The first character of the predicted list must be a digit or the letter "I"**.
  This ensures that the expression begins with a numeric value (e.g., 3, 112, etc.) or with the letter I in case of misclassifications of visually similar characters, such as: digit '1' erroneously classified as lower case letter 'l'
- **The last character of the predicted list is also a digit**.
  This condition enforces that the last predicted character in the list ends with a numerical value.
- **The prediction list must contain at least one character from the predefined set of valid variable electric symbols.**

  $$\mathbf{V} = \{V, R, C, L, I, i, B, u, U\}$$

  - V: voltage,
  - R: resistance,
  - C: capacitance,
  - L: inductance,
  - I: current,
  - B: for misclassified '3',
  - u, U: commonly misclassified voltage sources.
- **The prediction list must exclude invalid characters, specifically hyphens ($-$), decimal points (.), or the letter m character**.

Since none of those symbols are within the expected case to be handled in this condition, they are explicitly filtered out. This case handles only integer values for the numeric part. For the decimal numbers case, another condition was defined to handle this pattern.

For a better understanding, consider the predicted list illustrated in **Snippet 6.1**. The following list of predictions corresponds to the image **Figure 6.1**, where the characters in the list are in accordance with the rules presented in Condition 1, given that all the preceding criteria have been met.

Snippet 6.1: Example when the condition 1 is met.

```
[(('1', (1, 2, 20, 36)), (('V', (34, 1, 24, 39))
(('1', (45, 1, 28, 42)), (('0', (50, 1, 32, 45))]
```

Figure 6.1: Image that generated the previous predictions.

Once the aforementioned conditions are satisfied, an additional inference step is performed to correct previously detected errors. This refinement is implemented through the `make_new_prediction()` method, with adjusted parameters:

Snippet 6.2: New prediction function with hyperparameter.

```
new_pred = make_new_prediction(model, classes, image,
    predictions, order=0, var=1, pad=1)
```

- **var = 1** → modifies the fixed thresholding level in `cv2.threshold` during the preprocessing of bounding boxes to improve the object detection.
- **pad = 1** → reduces the padding around bounding boxes (default value is 2), thereby improving character recognition accuracy by minimizing excess background.

The new prediction generates a new list of predicted characters, which may match the first prediction or may reveal a new character that had previously been misclassified or omitted. This new prediction is then passed to the function `pp_prediction_cases()` responsible for organizing the predicted characters according to the context defined for that case.

If the new prediction contains a decimal point (.), the `pp_prediction_cases()` method processes the prediction list with the case set for **case=5**; otherwise, the

predictions are processed with the case set in **case=1** (subscription cases). The results of this function are detailed in **Section 6.5**.

CONDITION 2: Handle labels with scientific notation (passive components and dependent sources).

The purpose of this condition was to detect labels containing power expressions or scientific notation. The following label examples are pertinent to this specific case: $13.10^{-3}\quad i_6$ or $4.10^3 H$. In order for this to be achieved, the following conditions must be satisfied:

- **Both the first and last characters of the prediction list must be digits.** This approach facilitates the expeditious identification of dependent sources, i.e., those containing subscripts.
- **At least one character in the list corresponds to a recognized electric symbol, including `I`, `i`, `L`, `l`, `7`, `.`, `V`, `F`, `A`, `R`, or `H`, `Omega`,**. Letters 'L' and 'l' are included to take into account the misclassifications due to similar visual characters.
- **The prediction contains a sequence indicative of a superscript, such as `.10`, `.10-`, `m.10-`, or `m10-`.**

The following example illustrates a prediction list that satisfied condition 2.

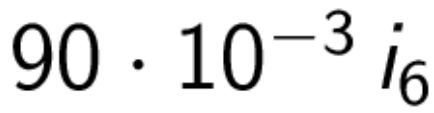


Figure 6.2: Images that generated the following list of predictions.

Snippet 6.3: Example when the condition 2 is met.

```
[('9', (0, 7, 21, 37)),
('0', (24, 7, 22, 37)),
('.', (59, 24, 12, 11)),
('1', (85, 7, 20, 37)),
('0', (108, 7, 23, 37)),
('-', (135, 14, 23, 4)),
('3', (160, 1, 17, 25)),
('l', (190, 18, 10, 27)),
('.', (194, 5, 11, 10)),
('6', (200, 26, 15, 25))]
```

Once the condition is satisfied, the prediction list is processed with **pp_prediction_cases()** function, using the hyperparameter `case=4` to handle superscript-related formatting for passive components and those containing dependent sources.

An example of this condition is provided in **Section 6.5.4**.

CONDITION 3: Labels containing rational values.

This condition is intended to identify annotations that contain rational numbers, such as $\frac{12}{25}H$ or $\frac{3}{5}I_5$. The initial prediction list must satisfy the following criteria to execute this condition:

- **The first character belongs to the set {`.`, `-`}.** The point (.) symbol has been included in order to handle a possible misclassification of a hyphen symbol that occurred in the initial prediction.
- **At least one subsequent character in the prediction list corresponds to a valid symbol from the set {`i`, `l`, `L`, `v`, `omega`, `I`, `V`, `C`, `H`, `A`, `E`, `F`, `G`, `S`, `P`}.**

Snippet 6.4 illustrates an exemplar of the array of predictions that align with the aforementioned rules. These predictions are generated from the **Figure 6.3**.

Snippet 6.4: Example when the condition 3 is met.

```
1
2 [(9, (22, 1, 25, 39)),
3 (., (104, 14, 8, 8)),
4 (I, (93, 34, 14, 39)),
5 (1, (111, 47, 22, 38)),
6 (., (3, 49, 63, 4)),
7 (1, (9, 64, 22, 37)),
8 (7, (37, 64, 25, 38))]
```

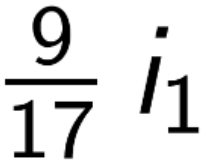


Figure 6.3: Image that generated the previous predictions.

When these conditions are satisfied, a new prediction is performed to verify that the characters are correctly classified, using the function presented in the **Snippet 6.5**.

Snippet 6.5: New prediction function with hyperparameter.

```
1 make_new_prediction(model, classes, image, predictions,
     order=1, var=2)
```

In this procedure, the parameter **`order=1`** specifies the order in which the bounding boxes are ordered. For instance, if the image contains the label $2/3\,F$, the

first prediction yields the sequence `-23F`, since the bounding boxes are sorted from left-to-right to top-to-bottom. By repeating the prediction with a different bounding box order (**top-to-right to bottom-left**), the classification sequence changes to `2F-3`. This adjustment is performed to facilitate the subsequent post-processing step, where the correct meaning of the label can be assigned more reliably.

The parameter `var=2` acts as a hyperparameter that controls the thresholding used during pre-processing of bounding boxes. In this case, the threshold is slightly reduced **(130)** compared to its default value **(158)**, which allows the method to generalize better and improve the accuracy of character classification, particularly in challenging or noisy cases.

Finally, the new prediction is passed to the `pp_prediction_cases(..., case=7)` function. In this case, the hyperparameter `case=7` triggers the execution of an additional method, `pattern_detection()`, which is further detailed in the **Section 6.5.5**. Basically, this method arranges the identified characters according to their correct fractional order and corresponding electrical component type (dependent sources or passive components). Likewise, the original first prediction was also passed to the `pp_prediction_cases` function to help maintain semantic formatting consistency.

CONDITION 4: Symbolic values for dependent sources (part I)

This condition refers to instances where labels contain subscripts with symbolic values, typically written with Greek or alphabetic letters for dependent voltages and current sources, such as $\mu_3 V_7$ or $\beta_2 i_{20}$, $r_2 i_{20}$. Please note that, in this particular instance, the symbolic values (β, α, ϕ) consist exclusively of a single digit as a subscript. The situation in which the subscript comprises multiple digits is addressed in Condition 5.

Formally, this condition is satisfied when the characters in the prediction list meet the following conditions:

- **The first character is in the Greek dictionary.** This dictionary contains the Greek letter names, for instance: mu, rho, alpha, etc.
- **The last character is a digit or alphabet letter.**
  Corresponding to the subscript cases.
- **At least one of the predicted characters belongs to the set**
  $V, I, i, g, r, \mu, \beta, L, l, U, u$.
  Some letters are included to manage possible misclassifications due to similar visual characters, such as ‘u’ instead of ‘v’, ‘l’ instead of ‘I’, etc. This set

comprises more Greek letter characters; however, for the purposes of simplicity in explanation, the set shown contains only two ('β', 'μ').

- **None of the predicted characters is a minus sign or hyphen (-).** Filtering out the hyphens avoids including fractional numbers, since these symbols are typically encountered in the prediction lists of the cases that contain those numeric values.

For better understanding, consider the following examples of prediction lists that satisfied this condition.

Snippet 6.6: Example when the condition 4 is met.

```
[('beta', (7, 30, 58, 49)),
('4', (74, 47, 31, 47)),
('I', (119, 30, 18, 49)),
('.', (133, 8, 8, 8)),
('1', (137, 48, 31, 45)),
('0', (174, 48, 31, 45))]
```

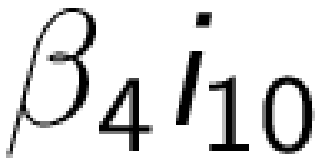


Figure 6.4: Image that generated the previous predictions.

Once the rules are satisfied, the prediction list of characters is passed to the function `pp_prediction_cases(..., case=9)` that will assign the context of the label based on the characters detected. In this method, the case indicated was (`case=9`), which ensures that subscripts are properly aligned and consistently formatted with respect to their base symbols (Greek or alphabetic letters). The **Section 6.5.7** delineates the operational dynamics of this case.

CONDITION 5: Symbolic values for dependent sources (part II)

The logic of Condition 4 is extended in this new rule. For this case, it is intended to match the labels whose symbolic values contain more than one digit as subscripts, such as cases involving multiple numerical subscripts (e.g., α_{56} or $\beta_{45}i_{20}$). Formally, this condition is satisfied when:

- **The first character is not a digit.**
- **The second and third characters on the prediction list are digits.** Ensures the numerical subscripts.
- **The last character is also a digit.**

- **Length of the predicted list is greater than 2 digits.**
- **At least one of the predicted characters belongs to the set** $\{V, I, i, g, r, \mu, \beta, L, l, A\}$. This list can be modified to include additional characters for the symbolic values if necessary. It is intended to encompass the most likely cases.
- **None of the predicted characters is a minus sign (-).**

To facilitate a more profound comprehension of this matter, it is useful to consider the following example of prediction lists that satisfied this condition.

Snippet 6.7: Example when the condition 5 is met.

```
[('alpha', (7, 30, 58, 49)),
('3', (74, 47, 31, 47)),
('3', (109, 47, 31, 47)),
('I', (150, 30, 18, 49)),
('.', (164, 8, 8, 8)),
('2', (168, 48, 31, 45),)
('2', (205, 48, 31, 45))]
```

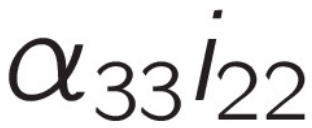


Figure 6.5: Image that generated the previous predictions.

When this condition is satisfied, a new prediction is carried out by passing the image to `make_new_prediction(..., order_contours=1)` function.

Snippet 6.8: New prediction function with hyperparameter.

```
new_pred = make_new_prediction(model, classes, image,
    predictions, order_contours=1)
```

In this step, the `order_contour` was set to 1, which resulted in the sorting of the bounding boxes from top-right to bottom-left during the **cv2.findcontours** method. Thereby refining the new prediction. Consequently, the new prediction list is passed to the function `pp_prediction_cases()` with hyperparameter values that depend on the first recognized character on the list.

DEFAULT CONDITION: General Case Handling

When none of the defined conditions are satisfied, the framework falls back to the default processing strategy. In this instance, all detected characters are merged without the application of any specialized rules or constraints. The merging process is executed via the `pp_prediction_cases()` function, which serves to consolidate the predictions into a unified textual representation, merging the predicted character with

the default settings. This default pathway is designed to ensure that even unclassified or unforeseen patterns are handled in a consistent manner, thereby preserving the interpretability of the recognition system.

Furthermore, the default condition was established with the aim of integrating the expression identified during the models initial prediction. This condition also plays a critical role in assessing whether additional conditions need to be defined or whether existing ones require a revision.

6.5 STEP 3: Post-processing Prediction Cases Method

The function `pp_prediction_cases()` acts as the core of the post-processing stage in the recognition pipeline. Its purpose is to refine raw predictions or the new prediction generated by the CNN-based model by enforcing domain-specific formatting rules, correcting systematic OCR errors, and ensuring consistency with physical notation in electrical engineering contexts.

The application of these rules is guided by the hyperparameter known as **`case`**. However, prior to a detailed examination of the rule-based cases, the inputs parameter of this method will be explained in detail.

6.5.1 Inputs:

The method receives the following parameters:

- **`data`**: List of predicted characters with associated bounding boxes.
- **`image`**: Input image containing the label expression.
- **`threshold_ratio`**: Scaling factor used to compute the margin around bounding boxes (default: 0.09).
- **`case`**: Control hyperparameter that determines which post-processing rule set to apply (linked to the conditions defined earlier).
- **`old_prediction`**: Optional string representing the first merged prediction (used for comparison in ambiguous cases).
- **`bbox`**: Optional bounding box information for refining special symbols (e.g., distinguishing between uppercase and lowercase v).

6.5.2 Algorithm

The function operates in three main stages:

STEP 1: Initialization

Once one of the conditions defined in the previous chapter is satisfied, the characters within the prediction list passed to the `pp_processing_cases()` function are concatenated into a raw merged string (merged_text) as illustrated in the **Snippet 6.9**. Additionally, a dynamic threshold (measured in pixels) is calculated based on the image dimensions. This threshold is subsequently used in **STEP 3** to expand the bounding box dimensions.

Snippet 6.9: Example of the characters merged from the list of predictions.

```
Prediction list: [('1', (1, 2, 20, 36)),('V', (34, 1, 24, 39)),('1', (45, 1, 28, 42)),('0', (50, 1, 32, 45))]
merged_text = '1V10'
```

STEP 2: Rule-based Normalization

Depending on the value of **case**, different regular expression patterns are applied to restructure the prediction into a compatible context format. In this section, some examples of the cases defined will be explained:

6.5.3 *case* = 1: SIMPLE SUBSCRIPTS SUCH AS $8I_2$ OR $3V_A$

When the hyperparameter **case** is set to 1, the function specifically handles simple subscript expressions, where the detected string structure corresponds to cases such as:

$2I_A$ (a number followed by a variable with a letter subscript) or
$2V_{12}$ (a number followed by a variable with a numeric subscript).

- **Pattern Matching**
  Pattern matching was conducted using Pythons **re** module, which provides functionality for working with regular expressions. This module enables the identification of specific patterns within text and supports the extraction of matched groups. To illustrate this process more clearly, the regular expression employed in this case was as follows:

  Snippet 6.10: Regular expression pattern for voltages and currents.

  ```
  pattern = r"(\d+)([VvIilL])\.?(\d+|[a-zA-Z]+)$"
  match_str = re.match(pattern, merged_text)
  ```

  This pattern in **Snippet 6.10** decomposes the string into three groups:

  1. **(\d+)** →**number** Match one or more digits (the base number, e.g., 2),
  2. **([VvIilL])** →**component** Match a single character that may represent voltage/current symbols (V, v, I, i) or possible misclassifications of `I` as `l/L`,
  3. **(\d+|[a-zA-Z]+)** →**subscript** Match either a sequence of digits or alphabetic characters (the intended subscript).

- **Standardization**
  If the merged string is matched with the predefined pattern, the correction of some misreading characters is performed. For instance, if the detected letter is misclassified as **l** or **L**, it is reassigned to **i** (to correct OCR misclassifications

of visually similar letters such as "l" and "i"). In this way, it handles the inconsistency. Additionally, the letter is normalized to lowercase, ensuring consistency and standardization in the type of circuit label before adding the subscript.

To provide a clearer picture of the procedure, the following example of a typical dependent voltage source label is provided:

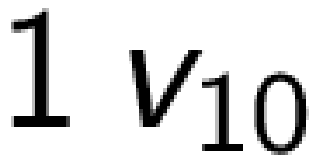


Figure 6.6: Dependent voltage source with Subscript format label.

After passing through the post-processing function, the predicted character list is merged into a single string like the following:

```
predicted_list = ((‘1’), (1, 2, 20, 36)),((‘V’), (34, 1, 24,
    39)),((‘1’), (45, 1, 28, 42)),((‘0’), (50, 1, 32, 45))
                processed_label = 1V10
```

In the current example, the string constitutes an unformatted sequence of characters. The `pp_prediction_cases()` method plays a crucial role in this context by assigning the appropriate semantic expression to the label, based on the specification of the `case` parameter. It is noteworthy that the resulting expressions are constructed in accordance with standardized labeling conventions commonly used in the representation of electrical circuits.

- **Reconstruction**
  After applying the regular expression depicted in **Snippet 6.10**, the corrected expression is then reconstructed in an equation style format. Since the `case=1` handles subscripts with one or more numeric or alphabetic characters, the final text label will be organized with the following form:

  ```
  processed_label = {number}{component}_{subscript}
  ```

  $$\texttt{processed\_label = 1V}_{10}$$

  As shown in **Figure 6.7**.

It is important to note that, in this case, the equation format used is consistent with the LaTeX format. However, it should be noted that not all cases defined in this section will conform to this format, since, following the assignment of the context to

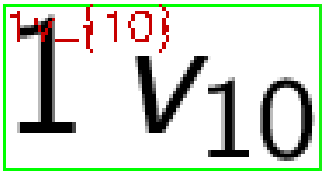


Figure 6.7: Dependent voltage source after postprocessing.

the label, the expected string will be passed to a function that will convert the entire expression into a correct LaTeX cleaner format.

6.5.4 *case* = 4: SCIENTIFIC NOTATION WITH SUPERSCRIPTS

When the hyperparameter **case** is set to 4, the function processes expressions involving superscripts, such as scientific notation or exponential scaling, which are frequently used in electrical variables, such as $2.10^{-3}V_A$, $18.10^{-6}F$. This case is addressed similarly to the previous condition.

In the superscript case, two possible label formats are identified. The first type contains dependent sources, such as V_1, I_4. The second type represents the passive components, for example, with numerical values and their component units, such as **V**, Ω, **A**, **F**, **H**.

TYPE 1: Dependent Sources (Numerical values using scientific notation).

Following the same line, consider the example described in **Condition 2**, where the predicted list is given by the following list:

```
predicted_list = [('9', (0, 7, 21, 37)),
        ('0', (24, 7, 22, 37)),
       ('.', (59, 24, 12, 11)),
        ('1', (85, 7, 20, 37)),
       ('0', (108, 7, 23, 37)),
       ('-', (135, 14, 23, 4)),
       ('3', (160, 1, 17, 25)),
      ('l', (190, 18, 10, 27)),
       ('.', (194, 5, 11, 10)),
      ('6', (200, 26, 15, 25))]
       merged_text = 90.10-3l.6
```

- **Pattern Matching**
  The regular expression to match this pattern of prediction is illustrated in **Snippet 6.11**:

Snippet 6.11: Regular expression pattern of a dependent source.

```
pattern = r'^(\d+(?:\.\d+)?)([-,+])(\d+)([iIlLvVRM(]|lm_lower)\.(\d+|[A-Za-z])$'
match_str = re.match(pattern, merged_text)
```

This pattern decomposes the detected string into:

1. **`(\d+(?:.\d+)?)`** →**`main_number`** The main number can be an integer or decimal, e.g., 9 or 9.10).
2. **`([-,+])`** →**`sign`** Capture the sign of the exponent, either - or +.
3. **`(\d+)`** →**`exponent`** Captures the exponent itself (e.g., 3 in 10^{-3}, 10^{3}).
4. **`([iIlLvVR(]|lm_lower)`** →**`component`** Captures a single character or token indicating the variable (common OCR confusions are normalized, e.g., I, l, L, i, (, or the placeholder lm_lower).
5. **`(\d+|[A-Za-z])`** →**`subscript`** The subscript attached to the variable can be numerical or alphabetic (e.g., A or 8).

- **Standardization**
  Once the pattern of the merged text has been matched, any character belonging to the set **`(l., L., (., lm)`** is normalized to the lowercase letter **`i`** to standardize the notation and mitigate inconsistencies arising from visual or classification ambiguities, particularly those involving the uppercase and lowercase of letter **`'I'`**, (when the stem of the 'i' is detected as 'l' or '('). Similarly, if the detected letter is **`'V'`**, it is replaced with lowercase **`'v'`** to maintain the notation.

- **Reconstruction and Result**
  Finally, **Figure 6.8** illustrates an example of how the matched string is reconstructed with the LaTeX expression as:

`merged_text = main_number`$^{\texttt{\{sign exponent\}}}$`component`$_{\texttt{\{subscript\}}}$

`Input:` `2.10-3VA` → **`Processed Output:`** $2.10\hat{}\{-3\}v_A$

`Input:` `90.10-31.6` → **`Processed Output:`** $90.10\hat{}\{-3\}i_6$

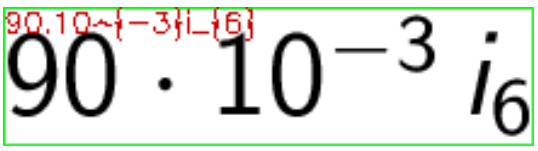


Figure 6.8: Dependent source with superscript values.

The final result of the string will then be reformatted with a cleaner LaTeX text style.

TYPE 2: Passive Component (Numerical values using scientific notation without subscript)

The second case involves numerical values that contain superscripts but not subscripts in the units. The following section will present the pattern used to identify these cases.

- **Pattern Matching**
  The regular expression is:

  Snippet 6.12: Regular expression for numeric labels with superscipts.

```
pattern = r'^(\d+(?:\.\d+)?)([-,+]?)(\d+)([vVRHAFS])$'
match_str = re.match(pattern, merged_text)
```

  Snippet 6.12 shows the pattern that captures strings that contain the following structure:

  1. **`(\d+(?:.\d+)?)`** → **`numeric_value`** the main number may be expressed as a decimal or an integer.
  2. **`([-,+]?)`** → **`sign`** Optionally captures the sign of the exponent.
  3. **`(\d+)`** → **`exponent`** Captures the exponent number.
  4. **`([vVRHAFS])`** → **`unit`** the associated component unit (e.g., V, R, H, A, F, S).

- **Standardization**
  After matching the structure, the standardization of the labels is implemented in accordance with the following structure.

$$\texttt{merged\_text = numeric\_value } {}^{\texttt{\{sign\}}}\ {}^{\texttt{\{exponential\}}}\ \texttt{unit}$$

  An example of this case is presented:

$$\texttt{Input: 8.10-3H} \rightarrow \texttt{Processed Output: } 8.10^{-3}H$$

6.5.5 *case* = 7: LABELS CONTAINING FRACTIONAL VALUES

The present case addresses the systematic identification and interpretation of fractional numerical labels in electrical circuits, particularly those that incorporate explicit units such as volts (V), amperes (A), farads (F), resistors (R), and others.

A method, titled `pattern_detection()`, has been developed to facilitate the post-processing of such labels. The method is structured into four principal analytical sections, each targeting a specific subset of the labeling patterns observed in circuit schematics. To comprehend the underlying logic of this function, please refer to the **Appendix A.1**, where the pseudo code is outlined.

The initial two sections of the `pattern_detection()` function are dedicated to the detection of controlled (dependent) sources, wherein the labeled values represent voltages or currents expressed as fractional multiples of other circuit variables. For instance, expressions such as $\frac{1}{2}v_1$, $\frac{5}{8}i_2$ are commonly used to denote voltage- or current-controlled sources. These elements are indicative of active components whose outputs are linearly dependent on a voltage or current present elsewhere in the circuit. These format corresponds to the standard forms of voltage-controlled voltage sources (VCVS), current-controlled voltage sources (CCVS), voltage-controlled current sources (VCCS), or current-controlled current sources (CCCS).

Whilst the third and fourth sections of the `pattern_detection()` method focus on fractional values associated with passive circuit elements, wherein the fractional number is paired with a unit (H,F,V,A) which is indicative of a component property. Examples of this include $\frac{14}{11}F$, which corresponds to a capacitor, or $\frac{7}{4}A$, which may represent either the rated current of a passive element or a measured value in the circuit context.

To differentiate between these cases, the `pattern_detection()` method employs regular expression (regex) matching for fractional patterns in conjunction with unit detection and contextual symbol analysis, similar to the previous cases explained. In this section, only 2 cases of this method will be explained to illustrate the idea. Those are dependent current sources (numerical values of current sources with a subscript) and a passive circuit element (numerical label with units).

In order to correctly identify the fractions, two predictions performed by the model were utilized. The first of these corresponds to the initial prediction (first prediction), with the standard preprocessing for bounding boxes employed for all the labels. The second corresponds to the new prediction performed in **Condition 3 (6.4.1)**, where some preprocessing parameters for BB detection were adjusted, to reorganize the order of the predicted characters to facilitate the organization further in this section.

Example 1: Dependent Current Sources Case in `pattern_detection()`

- **Regex for Fraction Detection**
  The core regular expression defined for the dependent current sources case is presented in the **Snippet 6.13**. This regex used to match the merged charac-

ters of the new prediction list (second prediction) performed in **Condition 3 (6.4.1)**:

Snippet 6.13: Regular expression example for the current dependent source.

```
match = re.match(r'^(\d+(?:\.\d+)?)[.]?([lLiI1])(\d+)
    [.-](\d+)$', merge_text)
numerator, token, subscript, denominator = match.groups
    ()
```

This pattern in **snippet 6.13** is broken down as follows:

1. **`(\d+(?:\.\d+)?)`** → **`numerator`** Captures the numerator of the fraction (integer or decimal).
2. **`[.]?`** → Optionally, it matches a literal dot, which indicates the dot on the lowercase "i", since OCR for cases with fractions detects characters from top-left to bottom-right order.
3. **`([lLiI1])`** → **`component`** Matches possible misclassifications of the current symbol (i), which could appear as lowercase L, uppercase I, or even digit 1.
4. **`(\d+)`** → **`subscript`** Captures the subscript of the current.
5. **`[.-]`** → **`division sign`** Matches either a dot or a hyphen, both of which are common OCR substitutions for the fraction separator /.
6. **`(\d+)`** → **`denominator`** Captures the denominator of the fraction.

- **Reconstruction of the Expression**

  The reconstruction of the standardized string is achieved through the expression of a dependent source. Preventing OCR errors from propagating into the final structured output.

Snippet 6.14: Standard form for the fractional cases.

```
label = "{numerator}/{denominator}i_{subscript}"
```

This guarantees the fraction structure is preserved. The current variable is always written with letter “i” standardized, even if OCR misread it as “l, L, or 1”.

To further clarify the proposed procedure, consider the following example. Assume that the merged string generated in accordance with the predicted list of characters of **Condition 3 (6.4.1)** is given by

```
predicted_list = [('9', (22, 1, 25, 39)),
                  ('.', (104, 14, 8, 8)),
```

```
('I', (93, 34, 14, 39)),
('1', (111, 47, 22, 38)),
('.', (3, 49, 63, 4)),
('1', (9, 64, 22, 37)),
('7', (37, 64, 25, 38))]
```

merged_text = 9.I1-17.

When applying the regular expression presented in **Snippet 6.15**, the resulting matching groups can be directly associated with the variables of interest, since each group was correctly matched:

$$\begin{aligned} \text{numerator} &= 9, \\ \text{token} &= .I, \\ \text{subscript} &= 1, \\ \text{denominator} &= 17. \end{aligned}$$

Based on this decomposition, it was possible to reconstruct the standardized form of the label, within the context of the current controlled source, obtaining the string in **Figure 6.9**. These strings will be converted into LaTeX syntax in a further stage of the `post_processing()` function:

$$9/17i_1$$

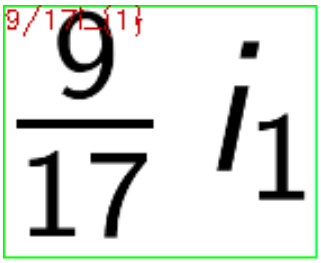


Figure 6.9: Controlling variable with fractional value.

- **Normalization using First Predictions**
  To ensure consistency with original model outputs (first prediction), an additional regular expression is performed on the `old_prediction` string:

Snippet 6.15: Regular expression example original model prediction.

```
match_old = re.search(r'(?:[.-])?(i_\{[\dA-Za-z]+\}|I_
  \{[\dA-Za-z]+\}|[\dA-Za-z]+)$', old_prediction)
```

The initial prediction made by the model, designated as **old_prediction**, is utilized in this method to rectify potential truncation or splitting errors. The purpose of this procedure is to verify whether the preceding prediction included a correctly formatted subscript, such as i_{10} or I_{23}.

In this context, when a match is identified, the subscript from the preceding prediction is utilized to correct potential truncations introduced during the optical character recognition (OCR) process. This mechanism is particularly relevant in scenarios where subscripts are inadvertently split into separate tokens. For example, consider the case in which the variable i_{10} is misrecognized by the model as two discrete tokens: i_1 and `0`. In such instances, the denominator of a fractional expression may be incorrectly interpreted as a continuation of the subscript, thereby leading to semantic ambiguity. To mitigate this, the method applies a rectification process wherein the isolated token is reattached to its corresponding subscript, resulting in the corrected form i_{10}. This ensures a clear distinction between subscripts and numerical denominators, thereby preserving the syntactic and semantic integrity of circuit element labels.

In a similar manner, in contexts involving other cases in the **pattern_detection()** function, such as voltage-dependent sources, or cases including passive component with unit of voltage, capacitor, inductor, or resistors with fractional expressions, the same logic as in the preceding example was implemented, utilizing regular expressions from the Python **re** library.

Example 2: Passive Circuit Elements with fractional values case in `pattern_detection()`

An additional example is presented for the passive circuit element with fractional values.

Snippet 6.16: Regular expression example of first model prediction.

```
1 match = re.match(r'^(\d+)([CHLAEFGSP|-]+)[.-](\d+)$',
      merge_text)
2 numerator, unit, denominator = match.groups()
```

Snippet 6.16 illustrates the regular expression used to match the complete merged string obtained from the prediction lists of characters in **Condition 3(6.4.1)** against the predefined pattern. This can be found in the fourth section of **Appendix A.1** procedure, which encompasses the cases corresponding to passive circuit elements. The following is a breakdown of the aforementioned regular expression:

1. **`(\d+)`** → **`numerator`** Captures one or more digits at the beginning of the string.
2. **`([CHLAEFGSP|-]+)`** → **`unit`** Matches one or more characters from the set {C, H, L, A, E, F, G, S, P, |, -}. This group is captured as the *symbol*, which

denotes the physical unit or element. Examples include: A (Ampere), L (Inductance), H (Henry), F (Farad), etc.

3. `[.-]` → **`division symbol`** Matches either a dot (.) or a hyphen (-), which serves as the separator. This marker plays the role of a division operator (analogous to writing a fraction, e.g., 20/7).
4. `(\d+)` → **`denominator`** Captures one or more digits.

Reconstruction

Snippet 6.17: Standard form for passive component with fractional cases.

```
label = "{numerator}/{denominator}{unit}"
```

Suppose the predicted merged string is given by:

```
predicted_list = [('1', (10, 4, 22, 38)),
                  ('4', (37, 5, 28, 37)),
                  ('H', (96, 15, 43, 60)),
                  ('-', (5, 51, 62, 6)),
                  ('1', (10, 67, 22, 38)),
                  ('5', (38, 68, 24, 38))]
merged_string = 14H-15
```

After applying the regular expression, the matching groups assign the corresponding values to each variable as follows:

$$\begin{aligned} \text{numerator} &= 14, \\ \text{unit} &= H, \\ \text{denominator} &= 15. \end{aligned}$$

Once each group is identified, the function to detect the correct order is applied by sending the characters matched in each group (denominator and numerator) to the `controlling_order()` function, see **Section 6.5.6**. This verifies the sequence of characters in the correct order, together with the standardized representation outlined in **Snippet 6.17**, yields the following final label expression:

$$14/15H$$

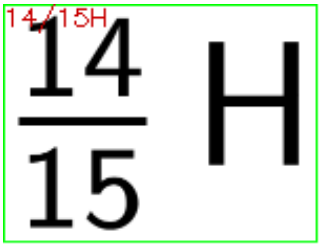


Figure 6.10: Passive Circuit Elements with fractional values.

6.5.6 CONTROL CHARACTER ORDER METHOD

The function **`controlling_order()`** has been designed to reconstruct structured numerical labels from a set of character-level predictions. Each prediction is denoted by a tuple comprising a character and its associated bounding box coordinates, stored in a list, e.g:

$$('5', (8, 3, 24, 38)), ('\Omega', (62, 11, 53, 62)), ('-', (4, 49, 33, 6)), ('3', (8, 63, 25, 40))$$

The primary objective is to separate and order the characters belonging to the numerator and denominator of a fractional annotation or label, ensuring that the sequence of symbols is sorted according to their spatial position.

Initially, the function identifies specific separator symbols (a dot or a hyphen) as tokens within the predictions list, and utilizes the vertical coordinate of their bounding boxes as a threshold to differentiate between the numerator and denominator regions. Given that this symbol is always located in the middle of the image containing the fractional value. Predictions that are located above this threshold are considered numerator candidates, whereas those that are located below or equal to the threshold are treated as denominator candidates.

To extract the characters of interest, the auxiliary function **`extract_sequence()`** is defined. This function sorts the predictions by their horizontal coordinate and iteratively selects characters that match the expected sequence (numerator or denominator). In this way, only characters consistent with the target string are retained, and their left-to-right order is preserved.

The procedure then constructs the numerator by means of concatenating the characters extracted from the upper region or upper bounding boxes. To avoid the reuse of characters, the indices of the selected characters are removed from the list of predictions. Consequently, the denominator is assembled from the lower region in an analogous manner.

The function returns two strings: the numerator and denominator, in a left-to-right order that has been standardized. The outputs of this process provide a robust reconstruction of the intended fractional label from the raw prediction data, thereby ensuring that both the spatial layout (above/below the separator) and the sequence order (left-to-right) are respected.

6.5.7 $case = 9$: DEALING WITH SYMBOLIC VALUES FOR DEPENDENT SOURCES

Finally, in the context of mathematics, Greek letters are conventionally employed to denote quantities, units, or variables. In the context of electronic circuits, these quantities, denoted by Greek letters, are symbolic variables whose values are determined by other circuit elements.

This case accounts for two possible scenarios that may arise when the prediction generated by the model is evaluated under **Condition 4**.

First Scenario:

The first situation occurs when the prediction misclassifies the digit '1' as the letter 'L' or 'l', triggering the rules established in **Condition 4**. For instance, consider the following merged string:

$$\textbf{Predicted string} = lll.5$$

However, the predicted string in the given example corresponds to a misrecognition of the numerical values of the expression. These were interpreted as the characters "l', 'L", instead of '11'. That is the reason why **Condition 4** is triggered when the expression includes either two consecutive alphabetic characters or one Greek letter. It is imperative that these potential misclassifications, which are a result of visual similarity between the characters, are addressed within the context of the case.

In this context, a special case was created to handle situations that correspond to numeric values with dependent currents and voltages. The regular expression **re.match()** was employed to ensure that the characters are arranged in the correct sequence. During the regular expression corresponding to the **'number'** field, any numeric values that were not correctly detected by the model are replaced with the correct number. For instance, the letter `"l"` or `"L"` is assigned to the number `"1"`.

The construction of the final label format follows the same procedure as the previously described cases. An illustrative example of this transformation is given below:

$$\textbf{Predicted string} = lll.5 \;\rightarrow\; \textbf{Final Label} = 11i_5$$

$$\text{number} = ll,$$
$$\text{letter} = l.,$$
$$\text{subscript} = 5.$$

Second Scenario:

The second case is intended for expressions that contain symbolic values, such as Greek letters and alphabetical variables. These annotations are typically rendered using subscript notation to represent dependent sources. In order to ensure the correct organization of the predicted characters, the function **'symbolic_detection()'** was created.

This method distinguishes among the possible types of labels in this case: those consisting exclusively of Greek letters, those combining Greek and alphabetic characters with subscripts, and those composed solely of alphabetic variables with subscripts.

For annotations containing Greek and alphabetic letters, the following rules are applied:

- If the predicted string begins with the name of a Greek letter (e.g., *mu*, *alpha*, *beta*, etc.), this prefix is extracted from the string.
- The remaining elements of the string are then compared against the regular expression, which is outlined in **Snippet 6.18**:

Figure 6.11: Dependent sources with symbolic values.

Example 1	**Example 2**
merged text = $beta4i10$	merged text = $mu2v7$
Greek letter extracted = *beta*	Greek letter extracted = *mu*
remaining part = $4i10$	remaining part = $2v7$

Snippet 6.18: Regular expression to match with the rest of the string.

```
1 match = re.match(r'([a-zA-Z0-9]+)([vViIlLuU1,])\.?([a-zA-Z0
    -9]+)', remaining_part)
2 subscript_symbool, component, subscript_component = match.
    groups()}
```

1. `([a-zA-Z0-9]+)` → **`subscript of symbolic letter`**. It is important to note that this corresponds to the number or letter following the Greek symbol, and that it captures the character or digit of the subscript.

2. `([vViIlLuU1,]).?` → **`component`** This expression captures the possible electrical components associated with the annotation, specifically voltages or currents, which are typically the dependent sources.

3. `([a-zA-Z0-9]+)` → **`subscript of alphabetic letter.`** Captures the subscript associated to the electrical component, with one or more digits.

If a match is found, two possible scenarios may occur:

Example 1	**Example 2**
geek letter = *beta*	geek letter = *mu*
subscript_symbol = 4	subscript_symbol = 2
component = i	component = v
subscript = 10.	subscript = 7.

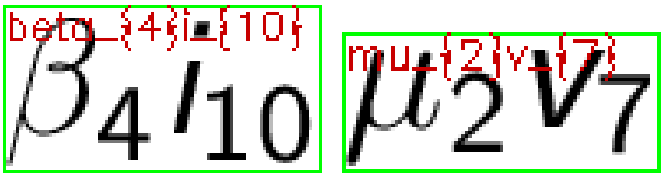


Figure 6.12: Dependent sources with symbolic values.

In instances where the annotation initiates with an alphabetic letter, the same aforementioned procedure is implemented. Initially, the first character of the sequence is extracted, after which the residual sequence is then compared against the regular expression in order to identify the values of each element that constitute the label. These characters are subsequently arranged into the designated label format. It should be noted that two scenarios are distinguished in this case: those containing a dependent source and those representing a single quantity. The final outcome is illustrated below.

Example 3
subscript_symbol = *vartheta*,
subscript = 2.

Figure 6.13: Dependent sources with symbolic values.

6.6 STEP 4: Bounding Box Consolidation

After reconstructing the predicted characters into a standardized label format, the bounding boxes are merged by identifying the outermost boxes and combining them into a single bounding box that encompasses the entire annotation elements. This procedure ensures a consistent and accurate representation of the bounding box for enclosing the formatted prediction within the image, as shown in the figures presented before **6.12,6.9,6.8**.

6.7 STEP 5: Outputs of Post-Processing phase

In addition, the prediction strings resulting from the `pp_processing_cases()` are stored in a text file as output. This file contains the predicted labels after assigning them a circuit context meaning in the post-processing phase **Snippet 6.19**. These results serves to monitor the predictions.

Snippet 6.19: Examples of results without converting to LateX cleaner format.

```
(5)
3/2A
(2)
(1)
(3)
6/17A
20/17H
8/5H
(0)
(6)
8/7A
5/18V
3/4V
11/6V
5/4omega
13omega
V_{9}
```

Another `*.txt` file was also saved, including the confidence levels associated with each raw prediction as shown in **Snippet 6.20**.

Snippet 6.20: Raw predictions with their confidence level

```
(5), confidence:1.00
3/2A, confidence:0.83
(2), confidence:1.00
(1), confidence:1.00
(3), confidence:1.00
6/17A, confidence:0.82
```

```
20/17H, confidence:0.80
8/5H, confidence:0.70
(0), confidence:0.88
(6), confidence:1.00
8/7A, confidence:0.81
5/18V, confidence:0.98
3/4V, confidence:0.82
11/6V, confidence:0.87
5/4omega, confidence:0.73
13omega, confidence:1.00
```

6.8 Weakness and Possible Improvements

The post-processing stage was designed based on the most common conventions for representing annotations in electronic circuits. As specific rules must be satisfied in order to reorganize the predicted characters correctly, this approach may not encompass all possible annotation cases. Consequently, if the final label generated by the application does not match the expected label despite the characters being predicted correctly, it is recommended that a new condition is defined for that specific case. If necessary, an additional case should also be incorporated into the `pp_prediction_case()` function.

It is important to note that the first prediction produced by the model determines which condition will be applied during post-processing. At this stage, an additional prediction can be made by adjusting the pre-processing parameters for the bounding boxes and making a new prediction. This ensures that each character or symbol is reliably detected. The post-processing function can then be applied by specifying a new condition or reusing an existing one. However, it is essential to verify that the newly defined condition does not overlap with any of the previously established cases.

Chapter 7

Testing Customized CNN Model vs Pix2tex

In accordance with the definition of the customized CNN model and the respective post-processing function, the subsequent step is to conduct a series of tests for the purpose of comparing the predictions generated by the Pix2Tex library and the predictions performed by the customized CNN with the post-processing stage.

Prior to the comparison, it was necessary to delineate an overview of the application utilized for the development of the electrical circuits recognition system. This comprises a concise description of the functions developed in a preceding study that were employed during the tests conducted. In a similar manner, this chapter delineates the methodology employed for the extraction of information and the management of errors.

7.1 Intelligence System Description

This work emerged from the necessity to adjust the predictions made by a predefined OCR tool called 'Pytesseract', which encountered difficulties in correctly detecting the annotations of an electronic circuit generated by the intelligent system developed in Cusano Michele's research thesis (2024). This intelligent system has been designed to detect all the components of a given electronic circuit image and extract the relevant information to solve the circuit. It is capable of recognizing nodes, components, the orientation of sources, text annotations, label components, topologies, and bridges, among other elements that may be present. For further details, refer to [27].

Similarly, all data collected from the circuit is stored in text files for future utilization. The final phase of image recognition is the process of graphic reconstruction. To this end, a versatile data structure was created for deployment through the MATLAB environment. The intelligent system was developed in Python; however, to regenerate the image once the detections were made, the information had to be transferred to MATLAB.

In order to accomplish this, two structures were required: the first, designated as G, comprises a simplified description of the circuit to be generated, incorporating rudimentary information regarding the nodes, corners, and components; and the second, G_{draw}, encompasses all the information about annotations, nodes, and edge coordinates, in addition to the bounding box of the entire circuit. These structures were utilized to establish a comparison between the original circuit and the circuit detected following the recognition task. The algorithm created for this purpose is described in Chapter 8 of the research thesis [27].

From this point, it was possible to perform the comparison between the predictions made by both methods, Pix2Tex and Customized CNN.

7.1.1 Modification of predefined Annotation function

Initially, the script '`annotations_ID.py`' developed by Cusano (2024) for the circuit annotations step was modified in this step. In this script, each annotation found in the circuit is extracted using the bounding boxes around the annotations. The resulting images are then saved in a directory associated with the generated circuit, following the standard format '`C_n_nodes_b_elements_yymmdd_hh:mm:ss`'. In this format, 'n' and 'b' are numbers configured during the creation of the circuit image, and they determine the maximum number of nodes and components that can be generated. In a similar fashion, the data pertaining to the evaluated circuit, including but not limited to the number of nodes, components, coordinates, and bounding boxes, JSON file structures of the circuits, is also stored in this folder.

Subsequent to the saving of the annotation images, the character recognition phase is initiated. For this thesis, the '`detect_annotations_with_contours()`' function of the '`annotations_ID.py`' script was adapted to incorporate the character detection using both methods, one using the Pix2Tex library and the other using the customized CNN model with the post-processing step developed in this work.

The results obtained were then saved in a text file containing the detected strings along with their bounding boxes coordinates, following the standardized format `text x y w h`, as shown in Snippet 7.1.

Snippet 7.1: Resulting prediction for both cases.

```
(3) 0.311276 0.902540 0.028560 0.033132
0.1\Omega 0.209708 0.102001 0.048270 0.023713
8mA 0.790695 0.817272 0.047466 0.023713
(0) 0.688992 0.351749 0.028560 0.033468
8.70\Omega 0.557254 0.737723 0.048807 0.025059
2.6H 0.354519 0.262866 0.048270 0.024891
v_{5} 0.412443 0.940128 0.019576 0.024218
0.46F 0.791164 0.437857 0.056718 0.024723
80mF 0.500201 0.437857 0.056449 0.024723
i_{13} 0.146621 0.581988 0.022660 0.027750
```

7.2 Comparison between Pix2Tex vs Customized CNN

In order to facilitate a comparison between the two methods, the baseline function '`Comparison.m`' was employed. The script, originally introduced by Michele Cusano in [27], was developed in the MATLAB programming environment for the purpose of completing the image recognition cycle. The procedure was implemented to automatically generate images of electrical circuits using the service *AutoCircuits* [1], which is a web application equipped with a MATLAB back-end that can be used programmatically to generate abstract circuit descriptions and the associated circuit diagram.

Subsequent to the image generation, the image recognition algorithm is initiated for the component and text identification, following the procedure outlined in the preceding section. The process thus reaches its conclusion with the regeneration of the circuit diagram based on the predictions obtained by the intelligent system that has been designed.

The script in reference is responsible for executing the steps delineated in **Algorithm 9**. The comparison was performed at six different levels, namely:

- checks the number of nodes

- checks the number of elements
- checks the distribution of node degrees
- checks the isomorphism of the graphs
- checks the isomorphism of the graphs and the preservation of edge labels
- checks the isomorphism of the graphs and the preservation of edge labels and values.

However, given the scope of this investigation, the most relevant check is the one relating to the values of the edge labels. If the values stored in the MATLAB structure 'G2' (Pix2tex) or 'G3' (customized CNN) coincide with those in the original structure, the prediction can be said to be correct. Conversely, if a discrepancy is detected, an error has occurred, and it is marked in a log file created for this scope.

Algorithm 9 Comparison Workflow

Step 1: Circuit generation
Generate circuits automatically.
Step 2: Intelligent system execution
Execute the designed intelligent system for image recognition.
Step 3: Structure creation
Generate G and G_{draw} structures from the JSON files generated.
Step 4: Structure comparison
Compare the original G MATLAB structure with the structure created from Pix2Tex prediction and customized CNN prediction (G_2 and G_3).

7.3 Errors Detection

The process of error detection occurs concurrently with the execution of the *Comparison.m* file. The creation of two text files is fundamental to the logging of results. The first file, entitled '`Log_file_execution.txt`', is used to record the information associated with the state of execution. The file contains the folder name of the circuit created, along with the state after execution, as illustrated in **Snippet 7.2**.

In this context, five distinct states can be distinguished:

1. No erros occurred.
2. Strutcture G_2 (Pix2tex) values differ from the original structure G, but structure G_3 (Customized CNN) values were correct.
3. Strutcture G_3 (Customized CNN) values differ from the original structure G, but structure G_2 (Pix2tex) values were correct.
4. Both structures (G_2 and G_3) differed from the G structure.

5. Error during execution of the Python script.

Snippet 7.2: Log execution file.

```
C_6_nodes_7_elements_20250928_141621.png: no errors occurred
C_6_nodes_7_elements_20250928_141730.png: no errors occurred
C_6_nodes_7_elements_20250928_141831.png: Python went wrong
C_6_nodes_7_elements_20250928_141940.png: no errors occurred
C_6_nodes_7_elements_20250928_142044.png: no errors occurred
C_6_nodes_7_elements_20250928_142202.png: no errors occurred
C_6_nodes_7_elements_20250928_142306.png: no errors occurred
...
C_6_nodes_7_elements_20250928_180602.png: both went wrong
C_6_nodes_7_elements_20250928_180719.png: no errors occurred
```

In a similar manner, the second file, entitled '`Errors.csv`', comprises a counter for five distinct cases. This file contains two columns, one indicating the case and the second with the counter, [**number, counter**] as shown in **Snippet 7.3**. The cases delineated in this file are analogous to the previous one, with the incorporation of two additional cases. The following list explains the meaning of each number.

1. Number of times the entire execution did not generate an error (circuits are identical)
2. Number of times both MATLAB structures G_2 and G_3 were not identical to the original structure G.
3. Number of times the MATLAB structure G_3 (Customized CNN) was not identical to the original one (circuits are not identical). While the structure G_2 (Pix2tex) was identical to the original G.
4. Number of times the MATLAB structure G_2 (Pix2tex) was not identical to the original one (circuits are not identical). While the structure G_3 (Customized CNN) was identical to the original G.
5. Number of times the Python execution failed.
6. Number of times the script has been executed.

Snippet 7.3: Errors counter.

```
1,324
2,1
3,2
4,0
5,4
6,331
```

If any of the aforementioned errors occur during the execution of the comparison '`Comparison.m`', the information pertaining to the circuit (images of the circuit, annotations, the circuit, the circuit nodes, the JSON structure, and `*.txt` files associated with component recognition and labels) is stored in a specific folder. This process facilitates access to the errors produced, thereby enabling subsequent analysis.

Once the information was collected in these two files, **Snippet 7.2** and **Snippet 7.3**, a script was created to separate the directories containing errors according to the categories mentioned above. Following the categorization of errors, an assessment was conducted to determine the number of images that generated errors within a circuit folder. For the purpose of comparison, the `*.txt` files (**Snippet 7.1**) for both Pix2Tex and CNN method created during the recognition of the circuit annotations (labels), were utilized. The predicted string labels for both methods were compared, with any discrepancies recorded as an error. From this point onward, the image that was misclassified can be more easily identified. It is important to clarify that, in **Snippet 7.1**, each string label represents an annotation on the circuit image.

The results obtained after counting the number of images misclassified in the circuits are saved in a `*.txt` file, as presented in **Snippet 7.4**

Snippet 7.4: Images misclassified per circuit.

```
Processing folder: pix2tex_went_wrong_cnn_no
Folder: C_6_nodes_9_elements_20250925_171741
Number of annotations: 26
Total errors: 1

Processing folder: cnn_went_wrong_pix2tex_no
Folder: C_6_nodes_9_elements_20250925_131348
Number of annotations: 30
Total errors: 1
Folder: C_6_nodes_9_elements_20250925_141446
Number of annotations: 27
Total errors: 1

Processing folder: Both_prediction_wrong
Folder: C_6_nodes_9_elements_20250925_125932
Number of annotations: 26
Total errors: 0
Folder: C_6_nodes_9_elements_20250925_142508
Number of annotations: 31
Total errors: 0

Processing folder: Python_errors
Folder: C_6_nodes_9_elements_20250925_120253
Number of annotations: 26
Total errors: 0
```

```
Folder: C_6_nodes_9_elements_20250925_125502
Number of annotations: 32
Total errors: 2
Folder: C_6_nodes_9_elements_20250925_140737
Number of annotations: 19
Total errors: 1
```

7.3.1 Process Pix2Tex output

The expression resulting from Pix2Tex is a mathematical LaTeX expression that primarily consists of a series of characters and commands commonly utilized in this format. In order to facilitate a comparison between Pix2Tex and the CNN + post-processing function, it is necessary to employ a more elementary text format for the predictions made, whilst preserving the structure of the expression, and the mathematical commands and symbols that may be generated by this OCR tool.

The **latex2text()** function was developed for the purpose of extracting all mathematical commands, characters, and superfluous spaces and font styles from the raw Pix2Tex expression. It receives the predicted label string with the mathematical LaTeX format, process the math string via regular expressions, and return a simplified, standardized "text-like" version of it with the minimal markup. Some details of the standardizations applied to clean expressions are mentioned below.

- **Cleans LaTeX formatting**:
  Removes spacing commands like `\;, \,, ~`, and unnecessary spaces.
  Deletes font/style commands (`\mathbf, \mathit, \textbf`, etc.).
- **Standardizes key math structures**:
  Detects and protects:
  Fractions cases (`\frac{...}{...}`)
  Subscripts cases (`_{...}`)
  Superscripts cases (`^{...}`)
- **Removes unnecessary braces and backslashes**:
  It deletes any extra `{}` outside the protected structures.
  Removes unnecessary `\left` and `\right`. To ensure these structures are not broken when braces are removed later.
- among others.

The following segment illustrates the functionality of the function:

Snippet 7.5: Example of transformation using the `latex2text` function

```
Input : \mathbf{\frac{11}{15}}~v_{6}
Output: \frac{11}{15}v_{6}
```

7.3.2 Testing Configuration

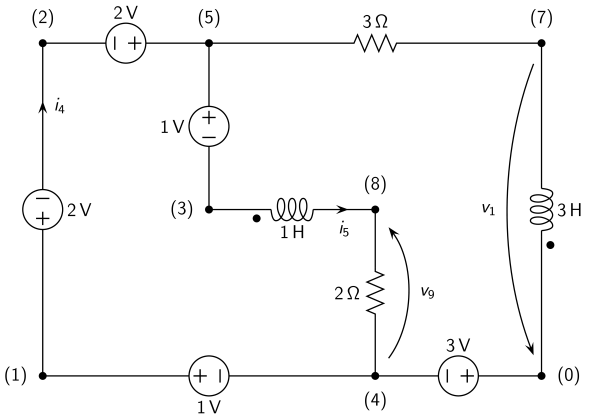
The performance of the Pix2Tex and Customized CNN methods was evaluated through three tests, which were executed simultaneously. The quantity of circuits generated is dependent upon the time that is set for execution. For first test, a duration of 90,000 seconds was established. For the second test, the execution time was set at 80,000 seconds.

The following section will provide a presentation of the results.

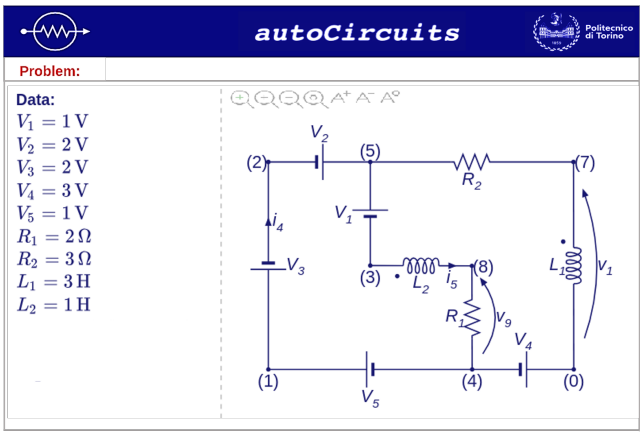
7.4 RESULTS

7.4.1 Correct Detections

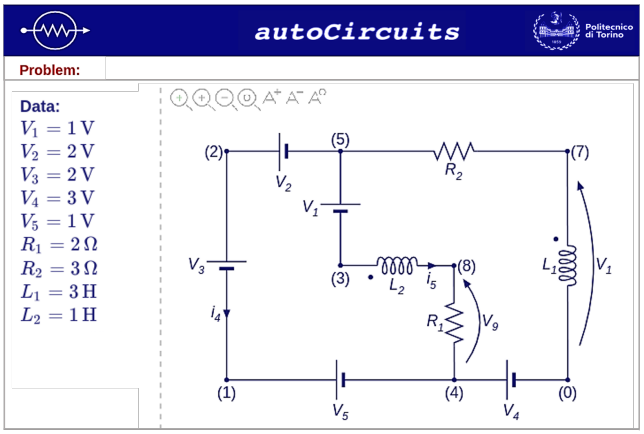
The following are illustrative examples of the circuits generated and correctly recognized for each of the two methods: Pix2Tex and CNN.



A)

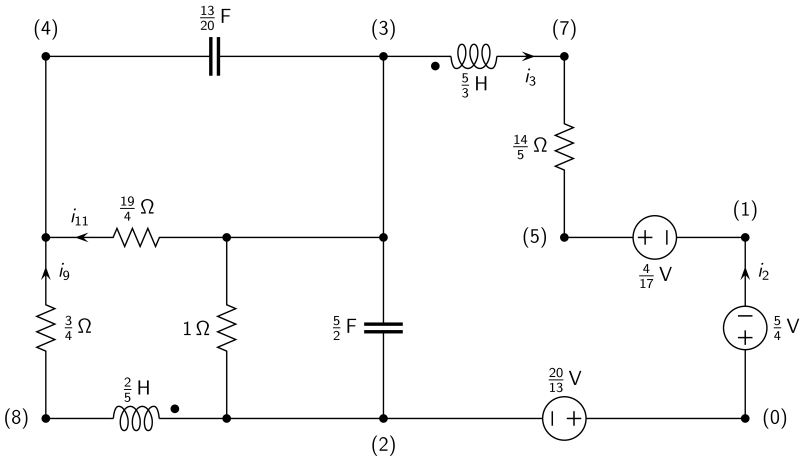


B)

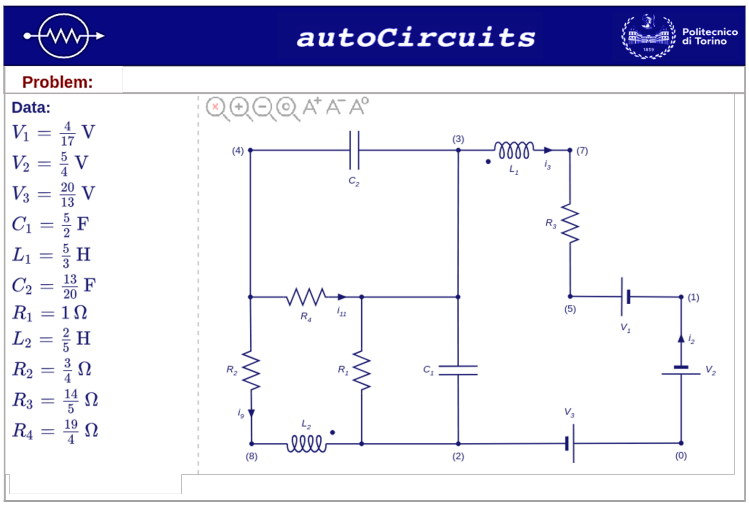


C)

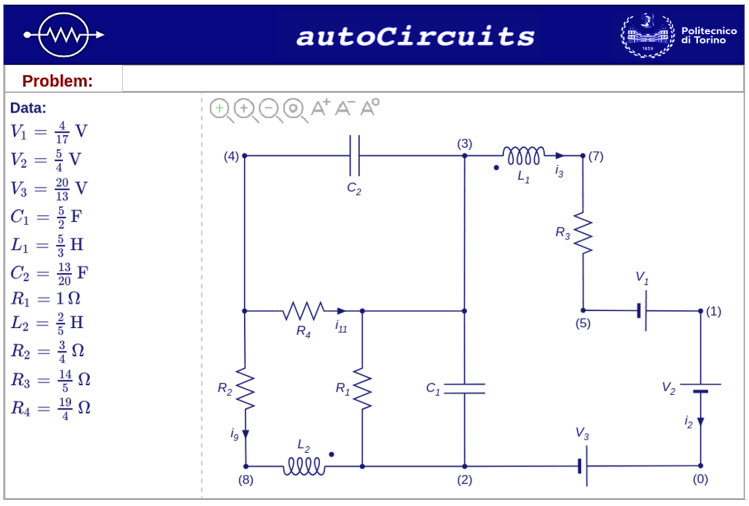
Figure 7.1: A) Original Circuit, B) Annotations predicted by Pi2Tex, C) Annotations predicted by CNN



A)

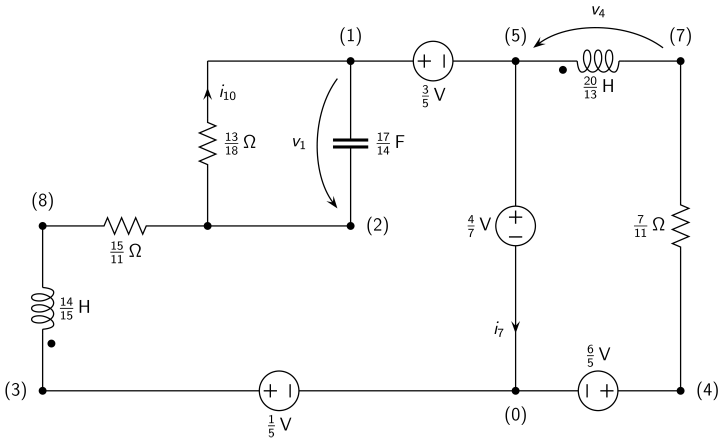


B)

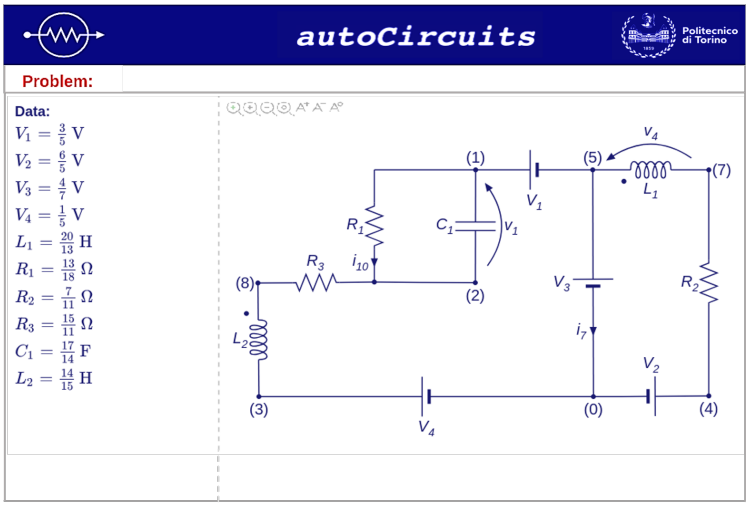


C)

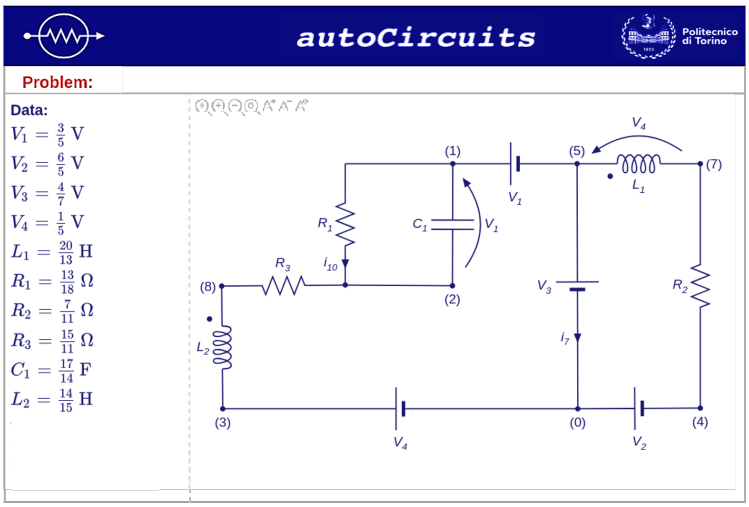
Figure 7.2: A) Original Circuit, B) Annotations predicted by Pi2Tex, C) Annotations predicted by CNN



A)



B)



C)

Figure 7.3: A) Original Circuit, B) Annotations predicted by Pi2Tex, C) Annotations predicted by CNN

The three illustrations presented in **Figures 7.1**, **7.2**, and **7.3** exemplify three distinct instances of a correct execution. As demonstrated in the electrical circuit diagram A), the numerical values assigned to the labels are consistent with the values predicted by both Pix2Tex and the customized CNN methods B) and C), thereby facilitating the reconstruction of the original image.

As previously stated, two tests were conducted in order to evaluate the recognition performance of the two methods. During the execution of these tests, the majority of the generated circuits demonstrated correctly classified annotations for both methods. However, it should be noted that in certain instances, errors were generated and classified into a single category. A comprehensive overview of the errors detected in these tests is provided in the following section.

7.4.2 TEST 1

	Total number of circuits	**Circuits without errors**	**Circuits with errors**
Total	1253	1190	63
%	100%	95.22%	4.78%

Table 7.1: Circuits statistics of Test 1.

The results obtained in Test 1 are displayed in **Table 7.1**. A total of 1253 circuits were generated, of which 1190 corresponded to circuits that did not report errors during execution, representing 95.22% of the generated diagrams. Meanwhile, 63 of the circuits, representing 4.78%, presented some of the errors classified into the following four categories:

(i) **Category 1**: The annotation predictions made by Pix2Tex correspond to the values of the original circuit. In contrast, the annotation prediction performed by CNN did not correspond to the values of the original circuit.

(ii) **Category 2:** The annotation predictions made by CNN correspond to the values of the original circuit. In contrast, the annotation prediction performed by Pix2Tex did not correspond to the values of the original circuit.

(iii) **Category 3:** Both predictions did not match the values of the original circuit.

(iv) **Category 4:** Errors generated during execution of the Python script.

As illustrated in **Table 7.2**, the errors generated in Test 1 are classified into the four categories aforementioned for the purpose of analysis. In **Category 1**, 6 errors were identified in instances where the predictions derived from the customized CNN were erroneous yet were accurately identified by Pix2Tex. Furthermore, column 3 of **Table 7.2** demonstrates the number of errors per annotation (label) of the circuit. For instance, consider a circuit containing 21 annotations, with only one misclassified

image label; the error per images is 1/21.

Error category	**Number of Errors per Category**	**Errors per number of images in the circuit**	**Type of Errors**
Category 1: CNN structure was not equal to the original G. Pix2tex structure was identical to the original G.	6	1/21 1/18 1/26 1/26 1/23 1/28	• 4 Errors due to image cropping. • 2 Classification error.
Category 2: Pix2tex structure was not equal to the original G. CNN structure was identical to the original G.	10	1/18 1/22 1/18 1/18 1/17 1/19 ...	• 8 Classification error. • 2 Errors due to the conversion to LaTeX cleaner format.
Category 3: Both predictions were wrong. Both structures differed from the original G.	17	2/23 1/20 1/18 1/23 0/25 1/14 2/19 2/25 1/18 1/21 ...	• Overlapping Labels. • Errors due to image cropping.
Category 4: Python Errors	28	N/A	• Overlapping Labels. • Errors due to image cropping.

Table 7.2: Summary of errors produced in Test 1.

In **Category 2**, 10 errors were identified, corresponding to incorrect detections by Pix2Tex, yet these were correctly identified by the Customized CNN. Of the 10 circuits that exhibited an error during the identification process, only a single annotation was misclassified with respect to the set of total labels allocated to that specific circuit.

Similarly, errors classified under **Category 3** accounted for 17 cases. The misclassifications are attributed to the fact that some annotations were not correctly identified during the process of label extraction by the Intelligent System, which employed the YOLO library. During the annotation detection and cropping stage, some bounding boxes extracted other nearby elements that affected label prediction.

Finally, **Category 4** indicates errors generated during the execution of the Python script. The intelligent system developed in [27] for the recognition of components, nodes, annotations, and the extraction of circuit information contained an error related to the libraries implemented for these tasks. Consequently, these represent propagation errors that have the potential to influence the result.

Visual representations of these errors are presented in the section 7.5.

7.4.3 Percentage of error considering each method separately

- Category 1

  Considering only the errors generated by CNN and assuming that those of Pix2Tex are correct, the percentage of errors during the classification from CNN is as follows.

$$Error\ per\ circuit\ using\ CNN = \frac{30 + 17 + 10}{1253 + 6} = 0.046 \approx 4.52\%$$

- Category 2

  Similarly, considering only the errors generated by Pix2Tex and assuming that those of CNN are correct, the percentage of errors during the classification from Pix2Tex is as follows.

$$Error\ per\ circuit\ using\ Pix2Tex = \frac{30 + 17 + 6}{1253 + 10} = 0.053 \approx 4.19\%$$

Prior to the execution of Test 2, modifications were implemented in the processing function `latex2text()`, which applies the conversion of Pix2TeX's raw predictions into a more refined, text-style format.

Some of these include: Adding a new regular expression to remove all commands related to font style, such as '`\mathbf`', '`\mathit`', and '`\textbf`', from the final predicted expression. Converting similar visual characters, such as 'I' misclassified as 'l', from the prediction. Additionally, some adjustments are required to clean up expressions containing subscripts with commands such as '`\textrm{...}`', which were not considered during the first test.

In a similar manner, the postprocessing function of CNN was modified due to the identification of a bug in the function during the initial test. It was observed that one of the conditions incorporated a parameter that was not being updated during execution. This resulted in the occurrence of errors during prediction.

7.4.4 TEST 2

	Total number of circuits	**Circuits without errors**	**Circuits with errors**
Total	962	943	19
%	100%	98.1%	1.9%

Table 7.3: Circuits statistics of Test 2.

Error category	**Number of Errors per Category**	**Errors per number of images in the circuit**	**Type of Errors**
Category 1: CNN structure was not equal to the original G. Pix2tex structure was identical to the original G.	2	1/19 1/21	• 1 Errors due to image cropping. • 1 Classification error.
Category 2: Pix2tex structure was not equal to the original G. CNN structure was identical to the original G.	1	1/21	• 1 Error due to the conversion to Latex cleaner format.
Category 3: Both predictions were wrong. Both structures differed from the original G.	4	1/24 1/23 1/22 1/21	• Overlapping Labels. • Errors due to image cropping.
Category 4: Python Errors	12	N/A	• Overlapping Labels. • Errors due to image cropping.

Table 7.4: Summary of errors produced in Test 2.

For the final experiment, the circuit generation time was increased. A total of 962 circuits were generated, of which 943 did not present any errors during execution or during the recognition of the circuit annotations. The error rate was found to be 1.9%, with only 19 circuits experiencing at least one error in the categories mentioned above, see **Table 7.3**.

As demonstrated in **Table 7.4**, a decrease in the number of errors produced in each category was observed in comparison to the preceding tests. This is attributed to the adjustments made between tests. The modification of post-processing functions in

the case of CNN, or the LaTeX conversion function in the case of Pix2Tex, facilitates the rectification of potential future errors. In addition, the nature of the errors observed in this study was comparable to those previously identified in Test 1.

7.4.5 Percentage of error considering each method separately

- Category 1

  Considering only the errors generated by CNN and assuming that those of Pix2Tex are correct, the percentage of errors during the classification from CNN is as follows.

$$Error\ per\ circuit\ using\ CNN = \frac{2+4+12}{962+1} = 0.018 \approx 1.86\%$$

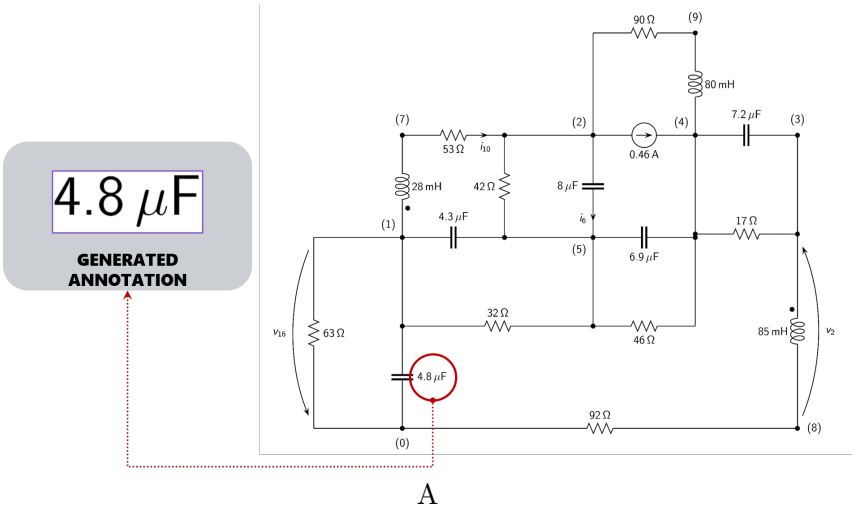
- Category 2

  If we consider only the errors generated by the Pix2Tex and assume as correct those of CNN, the percentage of error is the following.

$$Error\ per\ circuit\ using\ Pix2Tex = \frac{1+4+12}{962+2} = 0.017 \approx 1.76\%$$

7.5 Examples of Types of Errors Encountered

7.5.1 Category 1



A

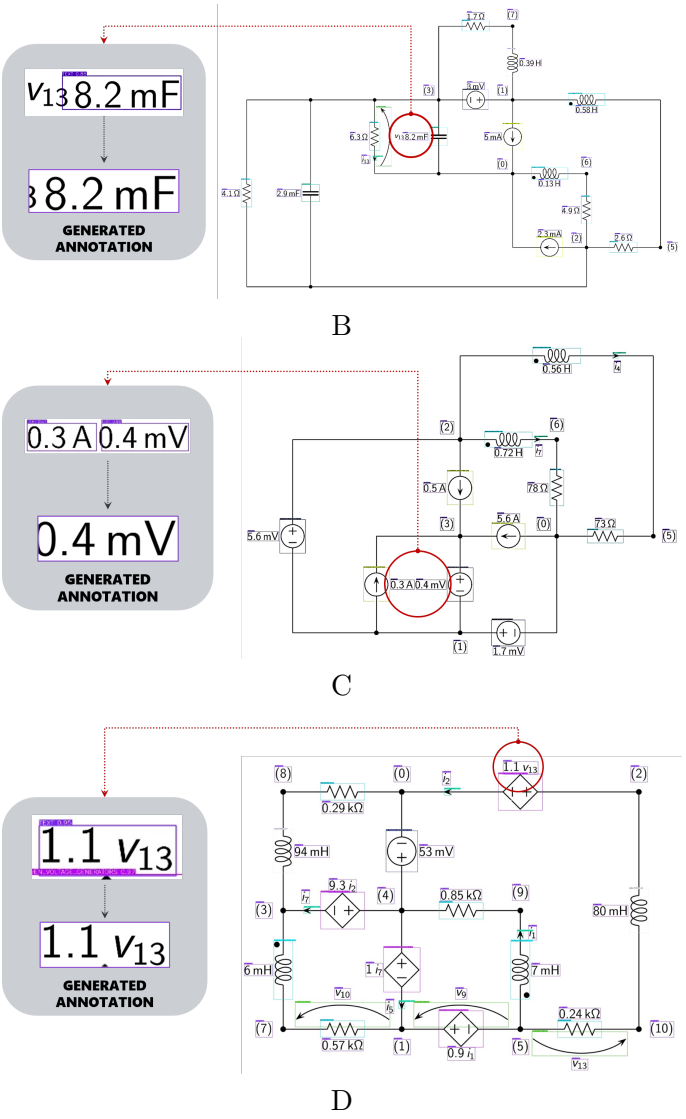


Figure 7.4: Examples of Category 1 errors.

Table 7.5: Errors detailed for Category 1.

Type of Error	Error per Annotation	Original Annotation	Detected by CNN	Detected by Pix2Tex	Image
Due to a misclassifications.	1/30	4.8μF	48μF	4.8μF	A
Due to the image cropping process.	1/21 1/18 1/30	8.2mF 0.4mV 1.1V_{13}	38.2mF).4mV 1.11V_{13}	8.2mF 0.4mV 1.1V_{13}	B C D

Table 7.5 provides a comprehensive overview of the errors identified in Category 1 during the experimental phase. As demonstrated in the table, only one image of the annotations of a single circuit was found to contain an error, due to two factors: character misclassifications or errors resulting from image cropping. Of the 2,000 circuit images that were assessed in the tests, a mere two errors were attributable to misclassifications for the Customized CNN, as evidenced in **Table 7.5**. The

residual errors were associated with the cropping process that was employed during the extraction of the annotations from the original circuit.

CNNs are more susceptible to misclassifications in the presence of any non-standardized shapes that may be in the image, whether due to an error during cropping of the annotation, clipping characters or overlapping labels. These erroneous characters are not predefined in the dataset utilized for the training of the network.

7.5.2 Category 2

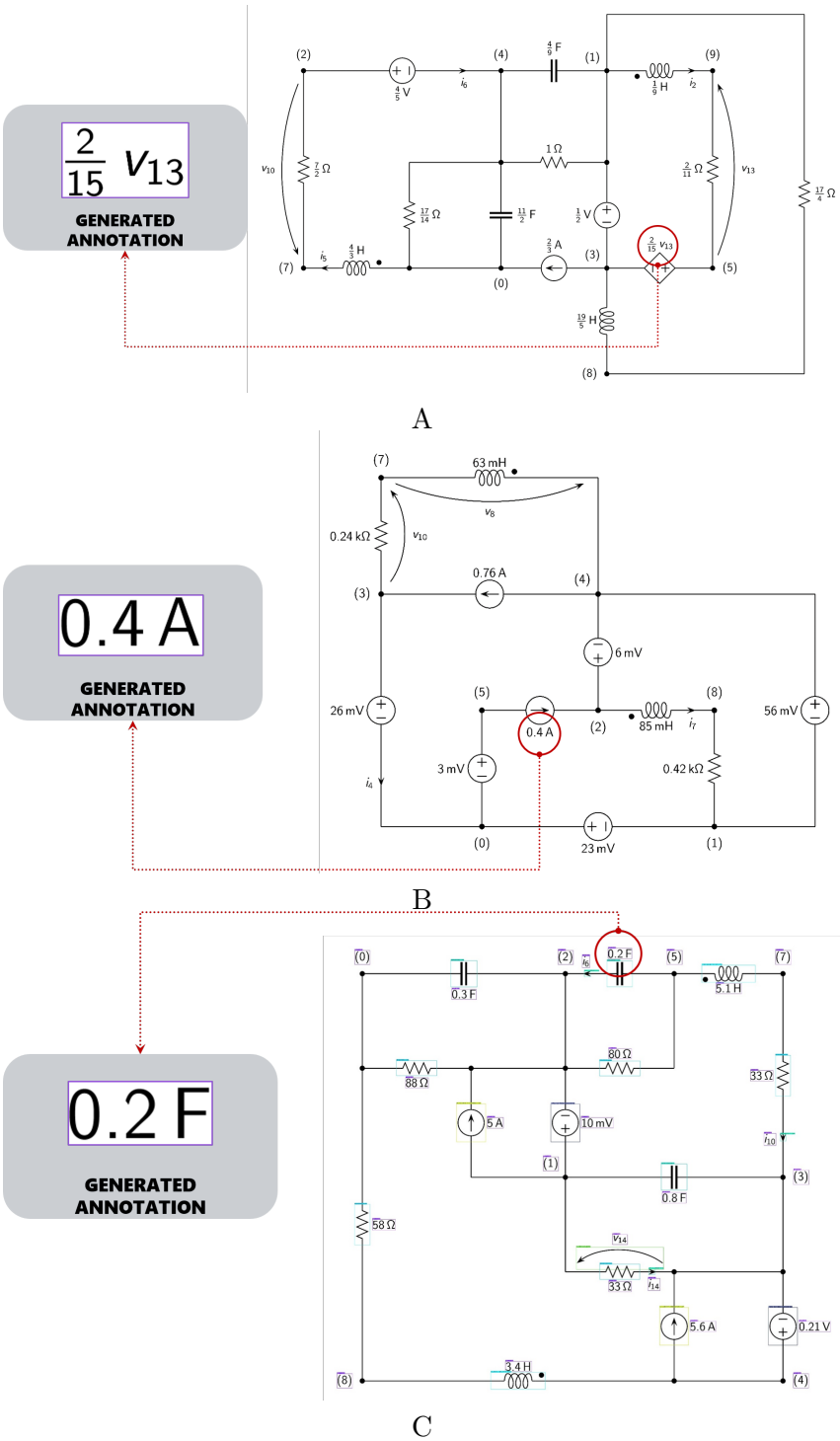


Figure 7.5: Examples of Category 2 errors.

Table 7.6: Errors detailed for Category 2.

Type of Error	Error per Annotations	Original Annotation	Detected by CNN	Detected by Pix2Tex	Image
Due to a misclassification	1/28	$\frac{2}{15}V_{13}$	$\frac{2}{15}V_{13}$	$frac215V_{13}$	A
	1/23	0.4A	0.4A	$omicron.4A$	B
	1/26	0.2F	0.2F	$omicron.2F$	C

Errors pertaining to Category 2 are predominantly attributable to misclassifications or arise during the conversion of the predicted strings to LaTeX format, as illustrated in **Table 7.6**. It has been observed that the function that converts the raw strings predicted by Pix2Tex to LaTeX format occasionally generates incomplete processing of the annotation string. These inconsistencies are the result of the application of a regular expression in order to clean up the raw output produced by Pix2Tex.

7.5.3 Category 3

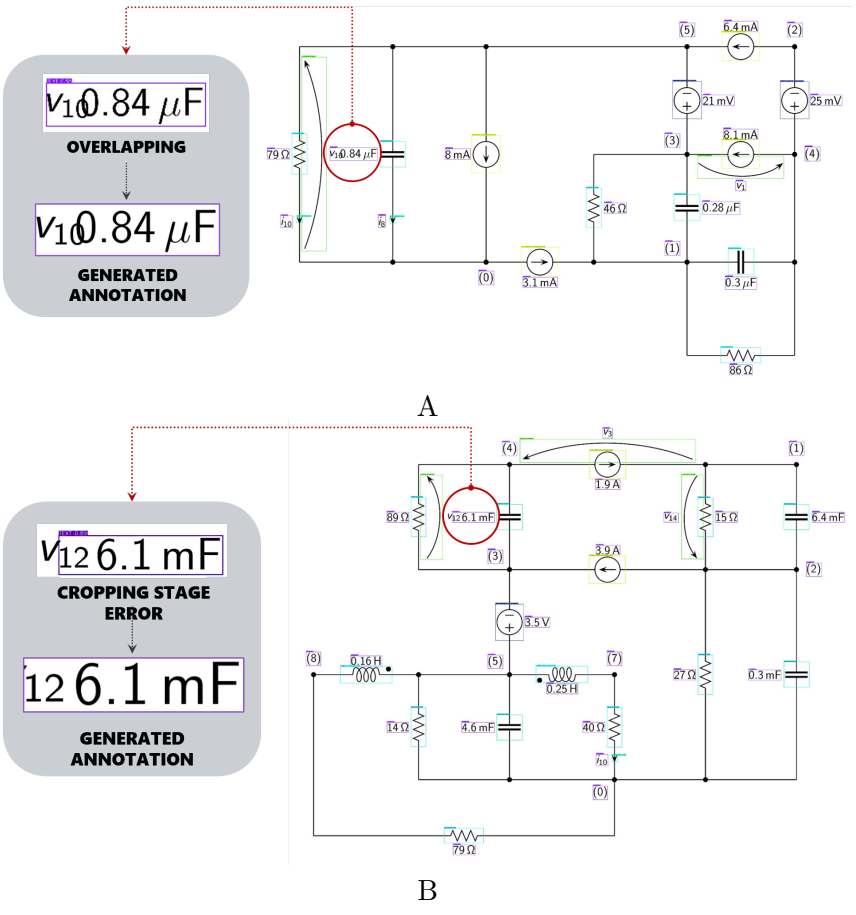


Figure 7.6: Examples of Category 3 errors.

In instances where both models failed during the circuit annotation predictions, i.e. in Category 3, the errors produced are primarily attributable to the annotation

extraction process. These errors are attributed to flaws in the Python YOLO library itself, which occasionally experiences difficulties in accurately identifying the label during detection when two labels are in close proximity to each other. This directly impacts the prediction of annotations, as these are the images that both models will subsequently evaluate. **Figure 7.6** depicts some examples of errors due to this category.

7.6 Recommendations for future works

The results demonstrated an optimal performance for the methods under evaluation. Errors in Categories 1 and 2 can be rectified by modifying the relevant functions. In the case of Pix2Tex, the function that converts the strings into text-style representations is applied subsequent to the prediction being produced. It is recommended that this function be adjusted in order to minimize the number of errors. Similarly, the adjustment of the post-processing function for the Customized CNN has been demonstrated to assist in the reduction of misclassifications of annotations.

Chapter 8

Conclusions

The motivation for this investigation stems from the necessity to rectify errors that occur during the process of detecting annotations in an electrical circuit. The intelligent system previously developed by a student utilized *Pytesseract* as an OCR tool for label detection. However, the reliability of this model for detecting the annotation was found to be deficient, with the model frequently generating erroneous detections.

In this research project, a convolutional model was implemented for the detection of characters comprising mathematical symbols, alphanumeric characters, and some letters of the Greek alphabet. The neural network was trained with a dataset that included images of these characters until the best accuracy was achieved. Subsequently, a post-processing function was applied to assign a meaning to the characters predicted by the CNN in order to standardize the format of annotations. A predefined Python tool, Pix2Tex, was utilized to detect mathematical formulas in images and convert them into text. In order to ascertain the most efficacious method for detecting circuit annotations, both approaches were executed concurrently during the generation of the circuit.

The preceding stages were successfully executed by an automated code capable of generating and testing images of electrical circuits. The findings from both methods demonstrated optimal performance in a series of tests conducted over a defined time period. It is evident that errors were identified during the prediction process. While these errors may not constitute the majority of the results obtained, they must nevertheless be taken into consideration when contemplating future deployment of the application.

Finally, it is important to mention that this application is a work in progress. It is recommended that handwritten character images be incorporated into the convolutional model, with the aim of creating a more robust model capable of detecting and identifying circuit labels from handwritten images.

Appendix A

Additional Functions

A.1 Pseudo-code for `pattern_detection()` function

This method is used during the postprocessing function of annotations containing fractional cases, comprising controlled sources or passive components.

Algorithm 10 Pattern Detection Function (Part 1)

Require: merge_text, old_prediction (optional), old_list (optional)

Initialize `text = ""`

—**SECTION I: Currents (`i, I`) —**

if pattern matches current fraction (e.g., `5.18-3`) **then**

    Extract numerator, token, subscript, denominator

    Refine values using `old_prediction`

    Reorder components with `controlling_order()`

    Format as `$numerator/denominator i_{subscript}$`

    return formatted text

end if

if pattern matches misclassified 'i' as ',' current (e.g., `4,7-5`) **then**

    Replace token (e.g., comma → I)

    Reorder and return `$numerator/denominator I_{subscript}$`

end if

if pattern matches simplified form, when point is detected first that the stem of the character 'i' (e.g., `9.1-54`) **then**

    Parse and reorder `controlling_order()`

    return formatted current expression

end if

if pattern includes subscript letter (e.g., `5.1A-3`) **then**

    Adjust subscript with `old_prediction`

    Handle 14-case and reorder

    return `$numerator/denominator i_{A}$`

end if

Algorithm 11 Pattern Detection Function (Part 2)

— **SECTION 2: Resistors (Ω)** —
if pattern matches resistor (e.g., `16omega-17`) **then**
    Extract numerator and denominator
    `controlling_order()`
    return `$numerator/denominator ω$`
end if
—**SECTION 3: Voltages (`V, v, U`)** —
if pattern matches voltage fraction (e.g., `3V-20`) **then**
    Normalize voltage symbol
    Handle missing or incorrect subscripts
    Fix wrong digits (`I/L/l  1`)
    Apply `controlling_order()` function.
    return `$numerator/denominator V_{subscript}$`
end if
—**SECTION 4: Other components (`R, L, C, A, F, H, etc.`)** —
if pattern matches generic component (e.g., `20L-7`) **then**
    Extract component symbol
    Apply `controlling_order()` correction and reorder.
    return `$numerator/denominator Letter$`
end if

Bibliography

[1] Stefano Grivet-Talocia and Politecnico di Torino. *autoCircuits: Automated Generation of Circuit Problems*. Accessed: 2025-10-09. 2018. URL: https://www.autocircuits.org/autocir_home.html (cit. on pp. 1, 88).

[2] Ayushi Chahal and Preeti Gulia. "Machine Learning and Deep Learning". In: *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 8.12 (Oct. 2019), pp. 4910–4914. ISSN: 2278-3075. DOI: 10.35940/ijitee.L3550.1081219. URL: https://www.ijitee.org/portfolio-item/L35501081219/ (cit. on pp. 5, 7–9).

[3] Koosha Sharifani and Mahyar Amini. "Machine Learning and Deep Learning: A Review of Methods and Applications". In: *World Information Technology and Engineering Journal* 10.07 (2023), pp. 3897–3904. URL: https://ssrn.com/abstract=4458723 (cit. on p. 5).

[4] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine learning and deep learning". In: *Electronic Markets* 31.3 (2021), pp. 685–695. DOI: 10.1007/s12525-021-00475-2. URL: https://doi.org/10.1007/s12525-021-00475-2 (cit. on pp. 5, 6).

[5] IBM. *What Is Deep Learning?* https://www.ibm.com/think/topics/deep-learning. Accessed: 2025-04-27. 2025 (cit. on pp. 6–9).

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://www.nature.com/articles/nature14539 (cit. on pp. 6, 7).

[7] Rana Sarker, H. M. Rasel, ABM Shafkat Hossain, and Md. Abu Saleh. *Integrating Climate Change Variables in Relative Humidity Prediction with Multivariate ARIMA and RNN Models*. Preprint. 2023. URL: https://www.researchgate.net/publication/376089842_Integrating_Climate_Change_Variables_in_Relative_Humidity_Prediction_with_Multivariate_ARIMA_and_RNN_Models (cit. on p. 7).

[8] DeepAI. *Distributed Representation*. https://deepai.org/machine-learning-glossary-and-terms/distributed-representation. Accessed: 2025-04-28. 2025 (cit. on p. 7).

[9] GeeksforGeeks. *NumPy: Types of Autoencoders*. Accessed: 2025-10-06. 2025. URL: https://www.geeksforgeeks.org/numpy/types-of-autoencoders/ (cit. on p. 8).

[10] Semiconductor Engineering. *Generative Adversarial Network (GAN)*. Accessed: 2025-10-06. 2025. URL: https://semiengineering.com/knowledge_centers/artificial-intelligence/neural-networks/generative-adversarial-network-gan/ (cit. on p. 9).

[11] A.R. Raut, A.S. Patil, and S.S. Patil. "Character Recognition using Machine Learning and Deep Learning - A Survey". In: *2020 International Conference on Smart Electronics and Communication (ICOSEC)*. IEEE. 2020, pp. 645–650. DOI: 10.1109/ICOSEC49089.2020.9215282. URL: https://ieeexplore.ieee.org/document/9167649 (cit. on pp. 10, 11).

[12] Haisam Abdel Malak. *7 Components of OCR to Master*. Accessed: 2025-04-25. Feb. 2025. URL: https://theecmconsultant.com/what-are-the-components-of-ocr/ (cit. on p. 11).

[13] Timea Bezdan and Neboja Baanin Dakula. "Convolutional Neural Network Layers and Architectures". In: *Proceedings of Sinteza 2019 International Scientific Conference on Information Technology and Data Related Research*. Belgrade, Serbia: Singidunum University, 2019, pp. 445–451. DOI: 10.15308/Sinteza-2019-445-451 (cit. on pp. 12, 13, 16).

[14] EITCA Academy. *What are the main components of a convolutional neural network (CNN) and how do they contribute to image recognition?* https://eitca.org/artificial-intelligence/eitc-ai-dltf-deep-learning-with-tensorflow/convolutional-neural-networks-in-tensorflow/convolutional-neural-networks-basics/examination-review-convolutional-neural-networks-basics/what-are-the-main-components-of-a-convolutional-neural-network-cnn-and-how-do-they-contribute-to-image-recognition/. Published by EITCA Academy, accessed October 14, 2025. Aug. 2023 (cit. on pp. 12, 13).

[15] Xia Zhao, Limin Wang, Yufei Zhang, Xuming Han, Muhammet Deveci, and Milan Parmar. "A Review of Convolutional Neural Networks in Computer Vision". In: *Artificial Intelligence Review* 57.4 (2024). Published online 23March2024, p. 99. DOI: 10.1007/s10462-024-10721-6. URL: https://doi.org/10.1007/s10462-024-10721-6 (cit. on pp. 14–18).

[16] Mohammad Mustafa Taye. "Theoretical Understanding of Convolutional Neural Network: Concepts, Architectures, Applications, Future Directions". In: *Computation* 11.3 (2023). Open Access; EISSN 2079-3197, p. 52. DOI: 10.3390/computation11030052. URL: https://doi.org/10.3390/computation11030052 (cit. on pp. 14, 17).

[17] Claudio Filipi Gonçalves dos Santos and João Paulo Papa. *Avoiding Overfitting: A Survey on Regularization Methods for Convolutional Neural Networks*. CoRR, arXiv:2201.03299. 27 pages; CCBY4.0 license. Jan. 2022. URL: https://arxiv.org/abs/2201.03299 (cit. on p. 17).

[18] Muhammad Syahid Rizky, Dede Luthfi Cahya, and Aryo Pinandito Wibawa. "Text recognition on images using pre-trained CNN". In: *arXiv preprint arXiv:2302.05105* (2023). URL: https://arxiv.org/abs/2302.05105 (cit. on p. 18).

[19] Paras Varshney. *VGGNet-16 architecture: A complete guide*. Kaggle Notebook. [Images]. 2020. URL: https://www.kaggle.com/code/blurredmachine/vggnet-16-architecture-a-complete-guide (cit. on p. 19).

[20] Lukas Blecher. *pix2tex*. https://pypi.org/project/pix2tex/. Accessed: 2025-08-07 (cit. on p. 20).

[21] Sciweavers. *LaTeX equation editor*. Accessed May 18, 2025. n.d. URL: https://www.sciweavers.org/free-online-latex-equation-editor (cit. on p. 23).

[22] T. E. de Campos, B. R. Babu, and M. Varma. "Character Recognition in Natural Images". In: *Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP)*. Lisbon, Portugal, 2009, pp. 273–280. URL: http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/ (cit. on p. 24).

[23] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. "The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes". In: *arXiv preprint arXiv:1708.01566* (2017) (cit. on p. 25).

[24] Luis Perez and Jason Wang. "The effectiveness of data augmentation in image classification using deep learning". In: *arXiv preprint arXiv:1712.04621* (2017) (cit. on p. 25).

[25] TensorFlow. *Keras: The high-level API for TensorFlow*. https://www.tensorflow.org/guide/keras. Last updated June 8, 2023. Retrieved from https://www.tensorflow.org/guide/keras. June 2023 (cit. on p. 31).

[26] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2015). URL: https://arxiv.org/abs/1412.6980 (cit. on p. 40).

[27] Michele Cusano. "Circuits and graphs recognition using Artificial Intelligence and Machine Learning techniques". Relator: Stefano Grivet Talocia. Master's thesis. Turin, Italy: Politecnico di Torino, 2024. URL: http://webthesis.biblio.polito.it/id/eprint/31028 (cit. on pp. 87, 88, 99).