OURE Report - 2023-2024
Machine Learning Circuit Component Recognition

Project Title: **Machine Learning Circuit Component Recognition**

Team Members: **Trenton Cathcart (tcgyq@mst.edu, EE)**

Advisor: **Dua, Rohit (rdua@mst.edu, MS&T EE)**

Department: **Electrical Engineering**

Date: **March 27, 2024**

Presented to the
Electrical & Computer Engineering Faculty of
Missouri University of Science and Technology
In partial fulfillment of the requirements for
OURE
April 2024 (SPS 2024)

Table of Contents

1. ABSTRACT

This project explores the integration of artificial intelligence (AI) into electrical engineering, focusing on developing a machine learning model for the recognition of circuit components in 2D schematics. By leveraging convolutional neural networks (CNNs), the study addresses the challenges of interpreting varied representations of circuit components, aiming to facilitate the learning process for beginners in electrical engineering. The approach involves a comprehensive preprocessing phase to optimize image data, followed by model training and refinement, including innovative techniques such as dynamic window sizing for improved component detection. The outcomes demonstrate the model's efficacy in accurately identifying circuit components, offering valuable tools for educational purposes and advancing the application of AI in electrical engineering. This contribution not only underscores the potential of machine learning in technical disciplines but also enriches the educational resources available to aspiring engineers.

2. INTRODUCTION

In the burgeoning field of artificial intelligence (AI), my fascination with understanding AI's foundational principles and their application to more complex models spurred me to embark on this research project. At the heart of my investigation lies the challenge of interpreting 2D circuit schematics, whether sketched by individuals or generated by computer-aided design (CAD) software. The objective was to develop a model capable of recognizing and identifying various common circuit components within these schematics accurately. This endeavor not only caters to my keen interest in AI's low-level workings but also seeks to bridge the gap to its higher-order applications.

Addressing this problem required a nuanced approach, given the intricate nature of circuit designs and the diverse ways components can be depicted. Historically, identifying and learning about circuit components has been a manual, often tedious process for beginners in electrical engineering. By leveraging machine learning, my project aims to significantly streamline this learning process. The model I developed is designed to discern and accurately label each component within a circuit schematic, thereby supporting novice electrical engineers in rapidly familiarizing themselves with various components. This application of machine learning not only aids in educational contexts but also highlights the broader potential of AI in foundational electrical engineering practices, a domain where such technological integration has been limited. Through this project, I endeavored to contribute to the enhancement of educational tools within electrical engineering and demonstrate the versatile applications of machine learning in simplifying and advancing our understanding of circuit components.

3. MAIN BODY

3.1. UNDERSTANDING PREPROCESSING AND NEURONS

3.1.1. PIXELS AND PREPROCESSING

In the pursuit of developing a neural network tailored for circuit component recognition, preprocessing of input data plays a pivotal role in ensuring both computational efficiency and model accuracy. As illustrated in **Figure 1**, two distinct images of a capacitor symbol are presented. The first image showcases the original, high-resolution depiction of the capacitor, providing a clear and detailed view. In contrast, the second image represents a resized version, scaled down to a resolution of 25x25 pixels. This resizing is a common preprocessing step, allowing for faster processing times while retaining essential features for recognition. Each pixel in this resized image carries an RGB (Red, Green, Blue) value, which is visibly annotated within the image itself. These RGB values serve as the initial input data for our neural network, translating visual information into numerical data that the model can process and learn from. By starting with such preprocessed images, our neural network is better equipped to efficiently recognize and classify various circuit components from raw visual data.

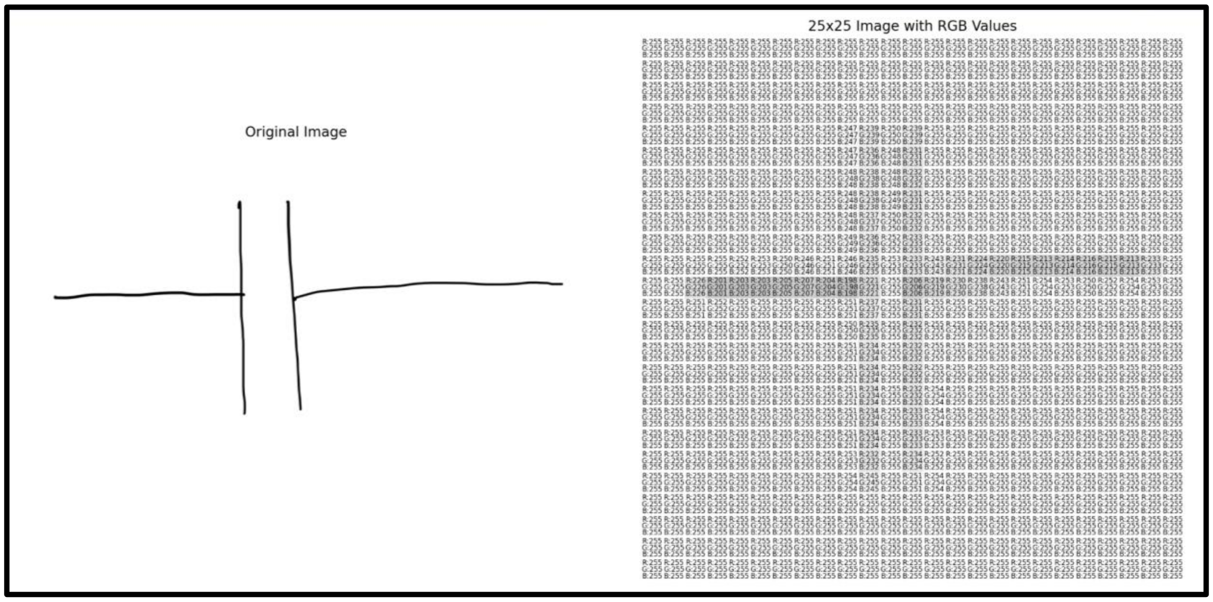


Figure 1: Understanding Pixels

3.1.2. **GRAYSCALE**

In many machine learning applications, especially in the realm of image processing for neural networks, simplification of input data can be paramount to achieving both computational efficiency and effective training. One such simplification technique is the conversion of colored images to grayscale. While colored images provide a rich representation with RGB values, they inherently possess a 3D structure (height, width, and color depth). On the other hand, grayscale images, as their name suggests, only capture shades of gray, effectively reducing the data dimensionality to a 2D structure (height and width). This transformation is particularly beneficial when the color information is not crucial for the task at hand, as is the case with our circuit component recognition. As depicted in **Figure 2**, a vibrant colored capacitor circuit symbol undergoes this transformation, resulting in a purely grayscale version. **Figure 3** further elucidates this process by presenting both the original and grayscale images side by side, with the grayscale values explicitly annotated within each pixel. By adopting this 2D representation, we significantly reduce the computational burden on the GPUs or CPUs, enabling faster processing times and more streamlined training of our neural network models.

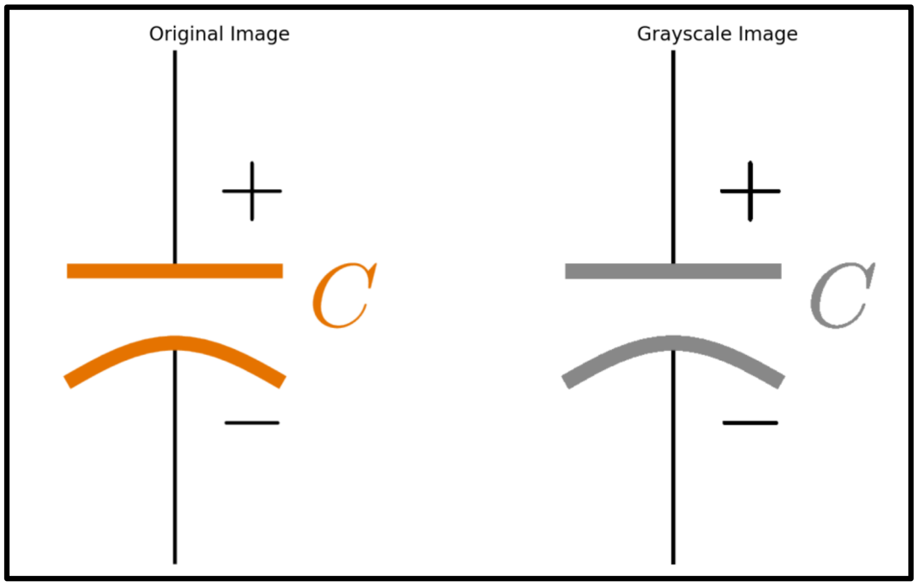
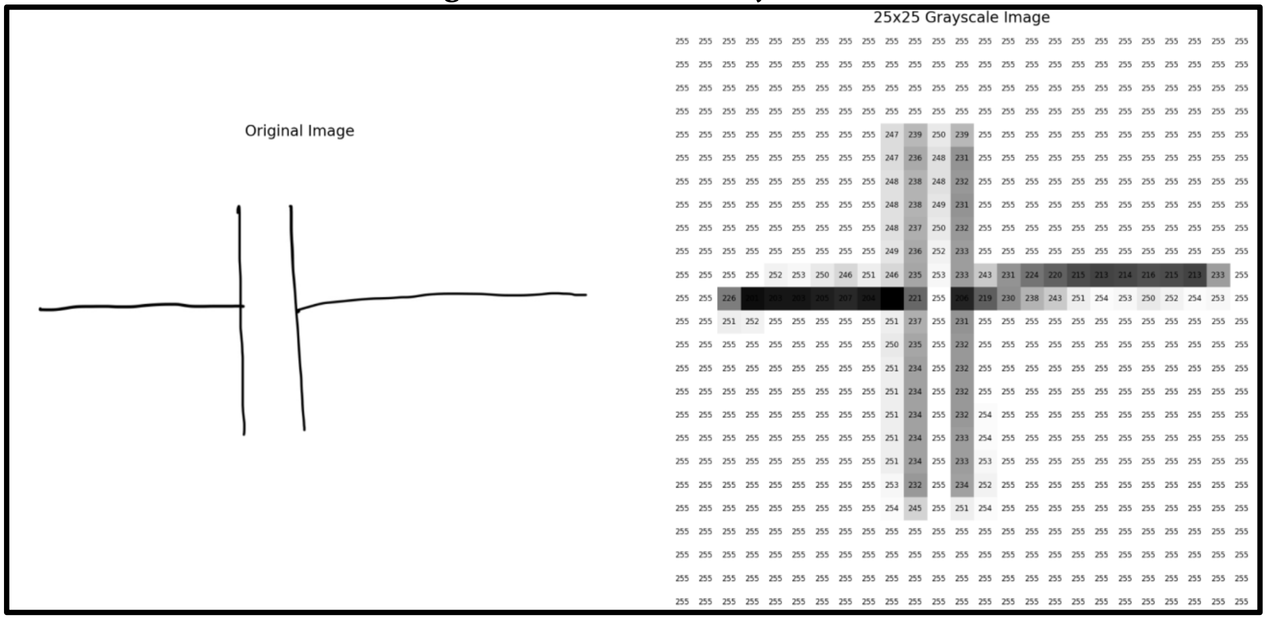


Figure 2: Colored to Grayscale

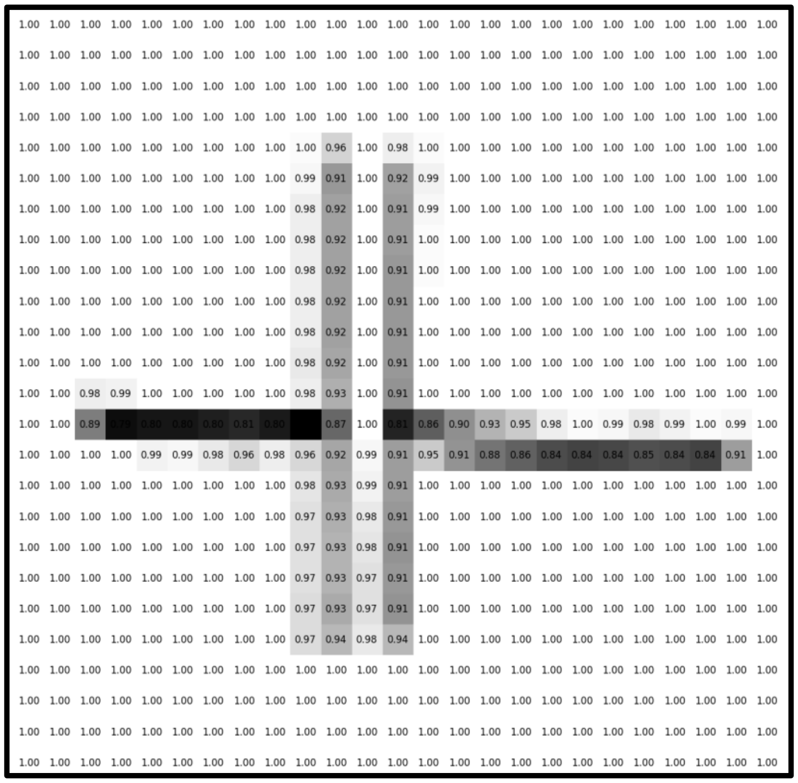


255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	247	239	250	239	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	247	236	248	231	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	248	238	248	232	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	248	238	249	231	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	248	237	250	232	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	249	236	252	233	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	252	253	250	246	251	246	235	253	233	243	231	224	220	215	213	214	216	215	213	233	255
255	255	226	201	203	203	205	207	204	[illegible]	221	255	206	219	230	238	243	251	254	253	250	252	254	253	255
255	255	251	252	255	255	255	255	255	251	237	255	231	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	250	235	255	232	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	251	234	255	232	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	251	234	255	232	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	251	234	255	232	254	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	251	234	255	233	254	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	251	234	255	233	253	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	253	232	255	234	252	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	254	245	255	251	254	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255

Figure 3: 2-D Representation

3.1.3. NORMALIZATION

Beyond the initial step of grayscaling, further optimization in neural network training can be achieved through the normalization of pixel values. Normalization, in essence, is the process of scaling input values within a specific range, typically between 0 and 1. This is especially beneficial for neural networks as it ensures that all inputs have a consistent scale, leading to a smoother and more stable optimization landscape. When pixel values range from 0 to 255, as is common in grayscale images, the variance in these values can lead to erratic and unpredictable updates during the training process. By normalizing these values, as shown in Figure 4, we not only simplify the computational task but also aid in the convergence of the model during training. The figure vividly illustrates the transformation of the capacitor's grayscale values, initially ranging from 0-255, to their normalized counterparts between 0 and 1. Each pixel in the image clearly displays its corresponding normalized value, underscoring the streamlined data representation that our neural network will utilize. This step, while seemingly simple, plays a pivotal role in enhancing the efficiency and performance of our circuit component recognition model. This output can be observed within **Figure 4**.



1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.91	1.00	0.92	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	0.98	0.99	1.00	1.00	1.00	1.00	1.00	0.98	0.93	1.00	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	0.89	0.79	0.80	0.80	0.80	0.81	0.80	[illegible]	0.87	1.00	0.81	0.86	0.90	0.93	0.95	0.98	1.00	0.99	0.98	0.99	1.00	0.99	1.00
1.00	1.00	1.00	1.00	0.99	0.99	0.98	0.96	0.98	0.96	0.92	0.99	0.91	0.95	0.91	0.88	0.86	0.84	0.84	0.84	0.85	0.84	0.84	0.91	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.93	0.99	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.93	0.98	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.93	0.98	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.93	0.97	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.93	0.97	0.91	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.94	0.98	0.94	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 4: Normalized Grayscale

3.1.4. NEURAL NETWORK BASICS

In the intricate world of neural networks, visual representations often serve as invaluable tools for understanding complex processes. **Figure 5**, provides a graphical representation of a neural network, showcasing interconnected circles (or neurons) and the weights associated with their connections. This interconnected web of neurons and weights forms the foundation of our network's architecture. Delving deeper into the mechanics,

Figure 6 elucidates how an image, in its raw form, undergoes transformation into the realm of linear algebra. A 25x25 image, for instance, translates to a staggering 625 neurons in the initial layer. Each pixel, or neuron, holds an "activation" value, representing its grayscale intensity, ranging from 0 (black) to 1 (white). As we progress through the network, subsequent layers also maintain activations normalized between 0 and 1. The interplay between layers is governed by weights, with activations in one layer influencing activations in subsequent layers. This intricate dance is captured in the weighted sum, where each pixel's activation is multiplied by its corresponding weight, culminating in a sum that encapsulates the collective information of all 625 neurons.

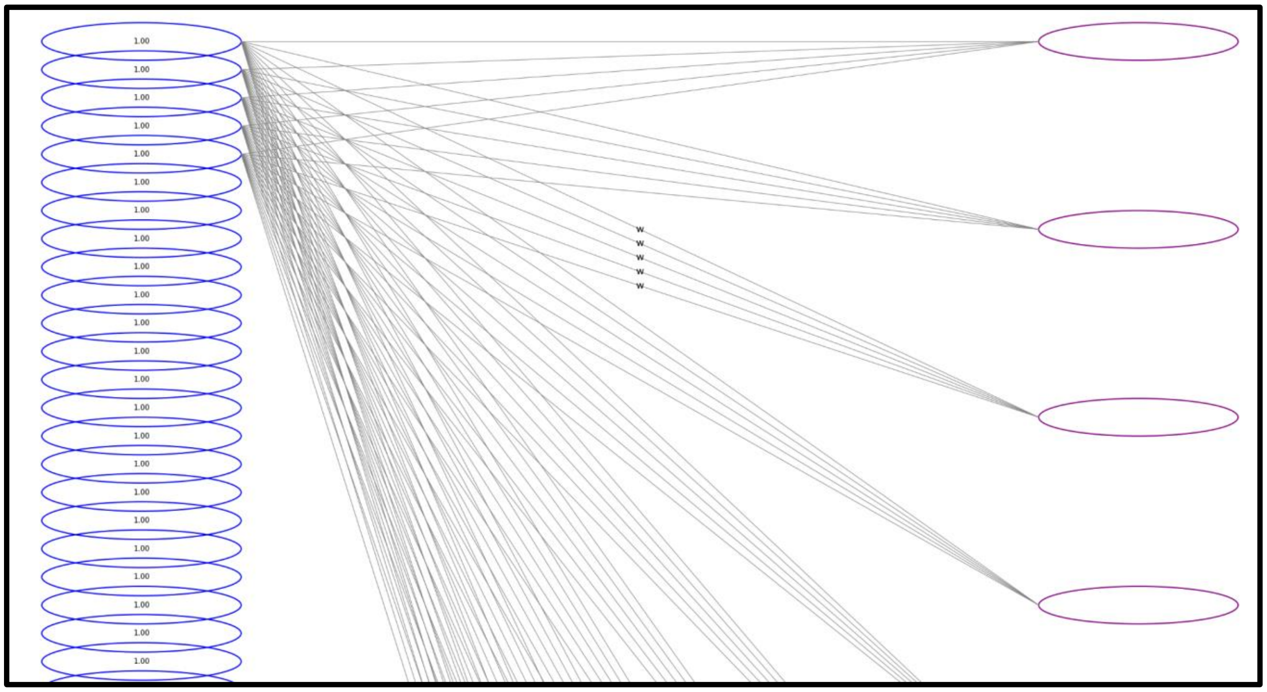


Figure 5: Neural Network Depiction

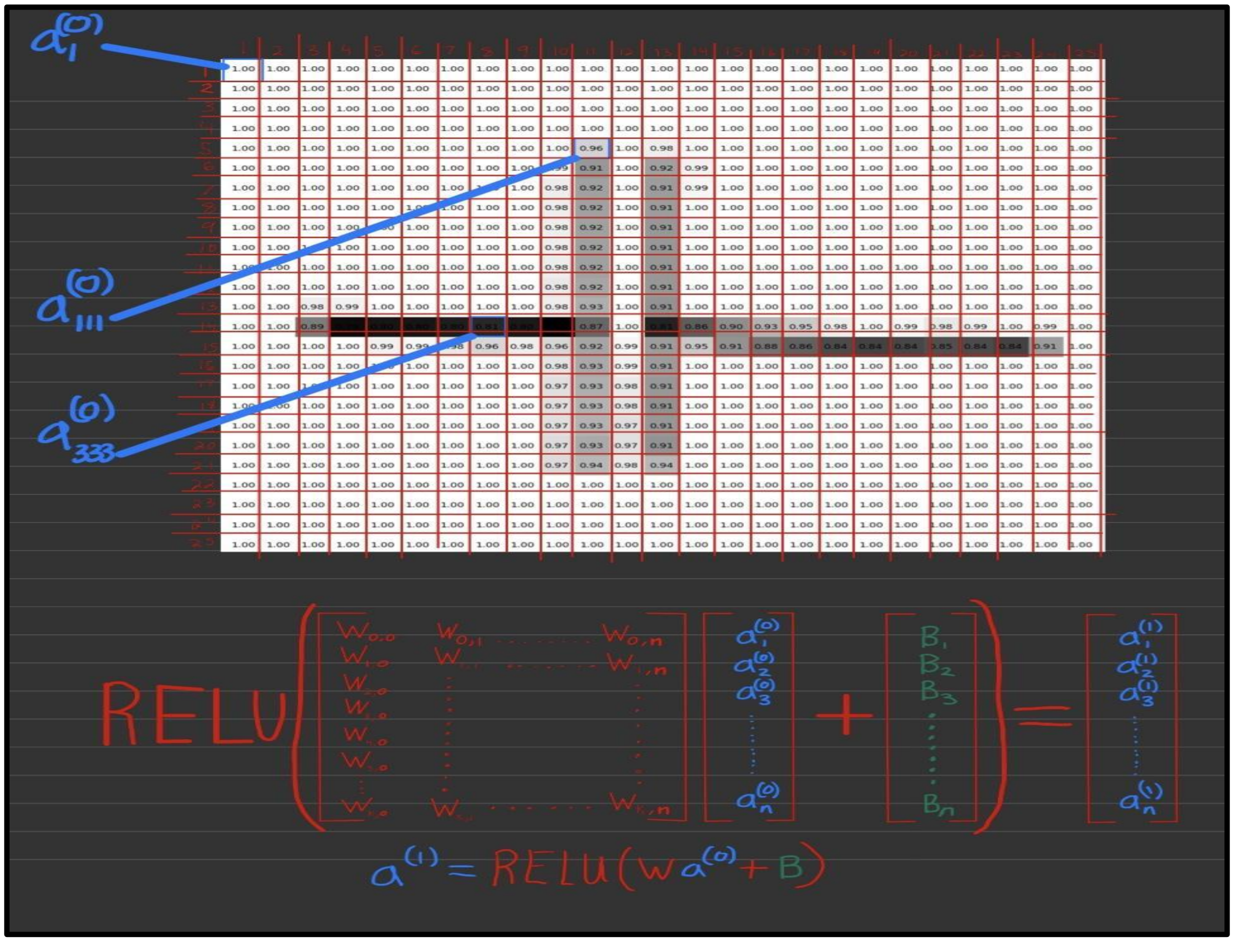

$a_1^{(0)}$
$a_{111}^{(0)}$
$a_{333}^{(0)}$
$\text{RELU}\left(\begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,n} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,n} \\ \vdots & \vdots & & \vdots \\ W_{k,0} & W_{k,1} & \cdots & W_{k,n} \end{bmatrix} \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ \vdots \\ B_n \end{bmatrix}\right) = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ \vdots \\ a_n^{(1)} \end{bmatrix}$
$a^{(1)} = \text{RELU}(Wa^{(0)} + B)$

Figure 6: Linear Algebra Image Depiction

3.1.5. ACTIVATION FUNCTIONS

However, to ensure our network's output remains normalized between 0 and 1, we introduce activation functions, as illustrated in **Figure 7**. Among the showcased functions - ReLU, Sigmoid, and Softmax - ReLU and Softmax have gained prominence in modern neural network architectures. ReLU, with its simple thresholding mechanism, offers computational efficiency and mitigates the vanishing gradient problem, a challenge often encountered with the sigmoid function. Softmax, on the other hand, is instrumental in multi-class classification tasks, providing a probability distribution over multiple classes. The sigmoid function, while foundational, compresses its input into a range between 0 and 1, making it particularly suitable for binary classification tasks. It ensures that even highly positive or negative weighted sums are squished into this normalized range. To further refine the activation, biases can be introduced, adjusting the weighted sum before it's processed by the activation function. This intricate combination of weights, biases, and activation functions orchestrates the symphony of computations that drive our neural network's prowess in circuit component recognition.

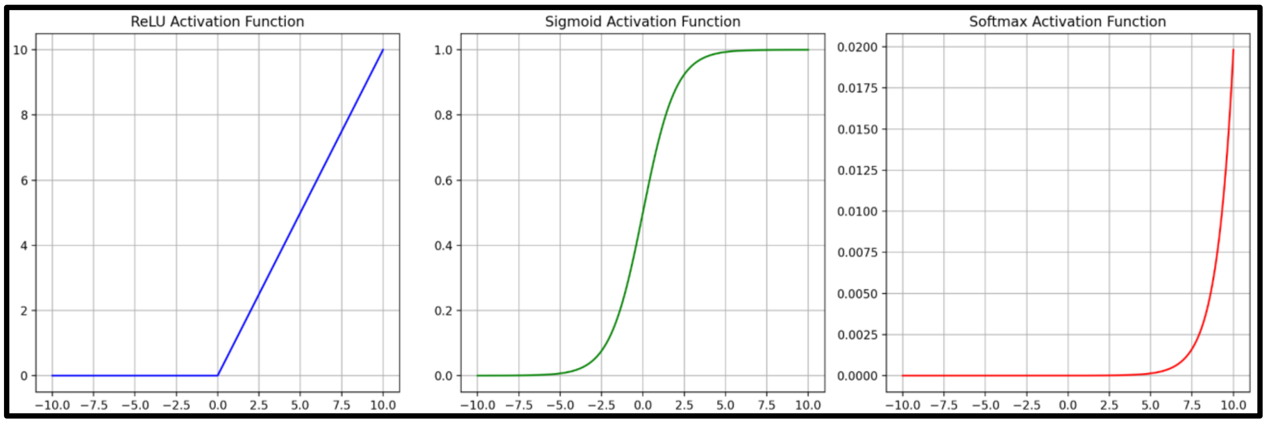
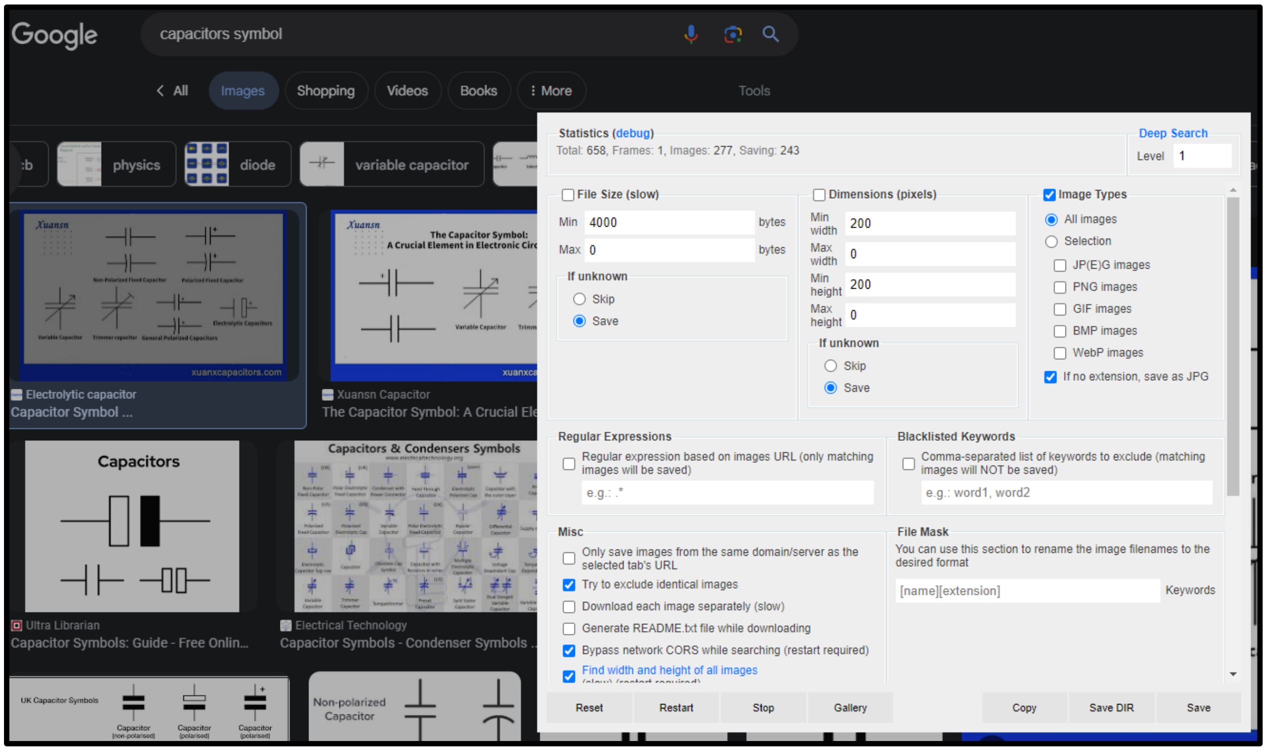


Figure 7: Activation Functions

3.2. BUILDING MODELS IN PYTHON

3.2.1. ACQUIRING DATASETS

The "Download All Images" Google Chrome extension serves as a vital tool in the initial stages of compiling a training dataset for machine learning models aimed at circuit component recognition. A user can conduct a Google search for specific circuit components, such as capacitors or resistors, and employ the extension to mass-download relevant images from the search results. The extension's built-in filters are particularly useful, allowing the user to refine the selection based on parameters like image resolution, file size, and format, ensuring that the downloaded images meet the necessary quality criteria for model training. This extension in action can be observed in **Figure 8.** After accumulating a substantial collection of images, I categorized the collected images into folders labeled “Resistor”, “Capacitor”,... i.e ... thereby creating a structured dataset, which can be observed within **Figure 9**.


Google
capacitors symbol
All
Images
Shopping
Videos
Books
More
Tools
physics
diode
variable capacitor
Electrolytic capacitor Capacitor Symbol ...
Xuansn Capacitor The Capacitor Symbol: A Crucial Ele
Capacitors
Capacitors & Condensers Symbols
Ultra Librarian Capacitor Symbols: Guide - Free Onlin...
Electrical Technology Capacitor Symbols - Condenser Symbols
UK Capacitor Symbols
Non-polarized Capacitor

Statistics (debug)
Total: 658, Frames: 1, Images: 277, Saving: 243

Deep Search
Level: 1

File Size (slow)
Min: 4000 bytes
Max: 0 bytes
If unknown: Skip [✓ Save]

Dimensions (pixels)
Min width: 200
Max width: 0
Min height: 200
Max height: 0
If unknown: Skip [✓ Save]

✓ Image Types
[✓ All images] Selection
JP(E)G images
PNG images
GIF images
BMP images
WebP images
✓ If no extension, save as JPG

Regular Expressions
Regular expression based on images URL (only matching images will be saved)
e.g.: .*

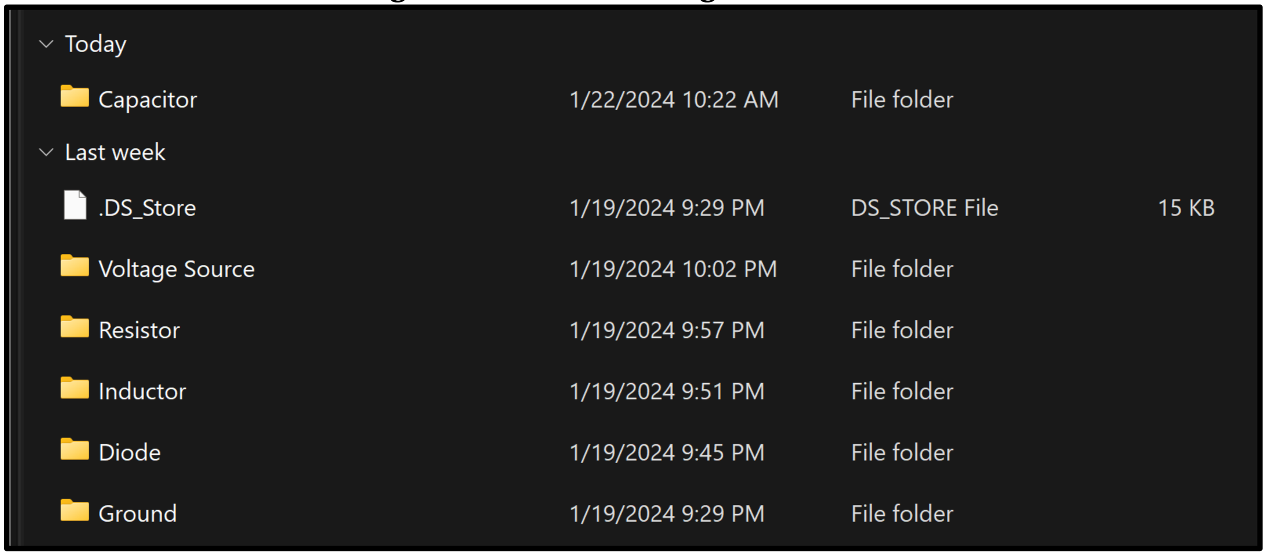
Blacklisted Keywords
Comma-separated list of keywords to exclude (matching images will NOT be saved)
e.g.: word1, word2

Misc
Only save images from the same domain/server as the selected tab's URL
✓ Try to exclude identical images
Download each image separately (slow)
Generate README.txt file while downloading
✓ Bypass network CORS while searching (restart required)
✓ Find width and height of all images (slow) (restart required)

File Mask
You can use this section to rename the image filenames to the desired format
[name][extension] Keywords

Reset | Restart | Stop | Gallery | Copy | Save DIR | Save

Figure 8: Download Images Extension



Today			
Capacitor	1/22/2024 10:22 AM	File folder	
Last week			
.DS_Store	1/19/2024 9:29 PM	DS_STORE File	15 KB
Voltage Source	1/19/2024 10:02 PM	File folder	
Resistor	1/19/2024 9:57 PM	File folder	
Inductor	1/19/2024 9:51 PM	File folder	
Diode	1/19/2024 9:45 PM	File folder	
Ground	1/19/2024 9:29 PM	File folder	

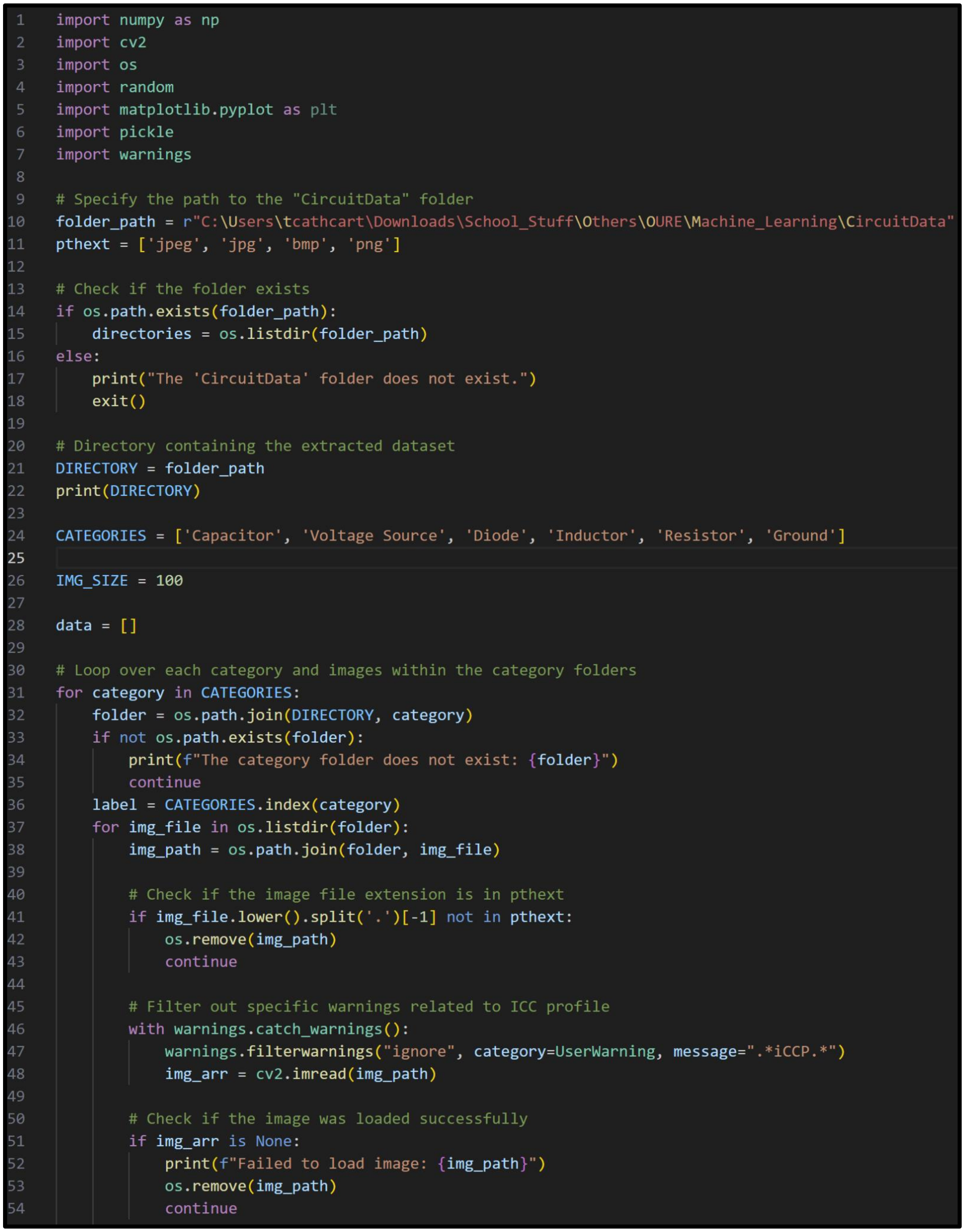
Figure 9: Sorted Images for Processing

3.2.2. EXPANDING AND PREPROCESSING DATA

The Python scripts 'Recolor', 'Multiply Data', and 'Image Circuit' form a comprehensive suite for image preprocessing, which is a crucial step in preparing a dataset for training a machine learning model for circuit component recognition. The 'Recolor' script is specifically designed to adjust the color properties of the images. This can include converting colored images to grayscale, which is a common practice for reducing computational load and focusing the model's learning on the structure rather than the color variations. Grayscale images help the model generalize better, especially when color isn't a distinguishing feature of the different classes.

On the other hand, 'Multiply Data' takes on the task of data augmentation. By creating modified copies of the original images through processes such as rotation, scaling, and flipping, the script enhances the dataset's diversity and size, which is essential for preventing overfitting and improving the robustness of the model. It ensures that the model is exposed to a variety of orientations and scales, mimicking the variance it will encounter in real-world applications. The 'Image Circuit' script complements these by resizing images to a uniform dimension, which is a prerequisite for training neural networks that require fixed-size input tensors.

Once the images are processed, they are ready to be compiled into a '.pkl' file format—a process often handled by Python's pickle library—effectively creating a serialized object that can be efficiently loaded during model training. Utilizing the powerful libraries of Keras and TensorFlow, the processed and serialized data then becomes the foundation for constructing and training convolutional neural networks. These networks are adept at capturing the hierarchical patterns in visual data, making them ideal for the task at hand—recognizing and classifying various circuit components based on their visual features. A typical preprocessing script can be observed within **Figure 10.**



```python
import numpy as np
import cv2
import os
import random
import matplotlib.pyplot as plt
import pickle
import warnings

# Specify the path to the "CircuitData" folder
folder_path = r"C:\Users\tcathcart\Downloads\School_Stuff\Others\OURE\Machine_Learning\CircuitData"
pthext = ['jpeg', 'jpg', 'bmp', 'png']

# Check if the folder exists
if os.path.exists(folder_path):
    directories = os.listdir(folder_path)
else:
    print("The 'CircuitData' folder does not exist.")
    exit()

# Directory containing the extracted dataset
DIRECTORY = folder_path
print(DIRECTORY)

CATEGORIES = ['Capacitor', 'Voltage Source', 'Diode', 'Inductor', 'Resistor', 'Ground']

IMG_SIZE = 100

data = []

# Loop over each category and images within the category folders
for category in CATEGORIES:
    folder = os.path.join(DIRECTORY, category)
    if not os.path.exists(folder):
        print(f"The category folder does not exist: {folder}")
        continue
    label = CATEGORIES.index(category)
    for img_file in os.listdir(folder):
        img_path = os.path.join(folder, img_file)

        # Check if the image file extension is in pthext
        if img_file.lower().split('.')[-1] not in pthext:
            os.remove(img_path)
            continue

        # Filter out specific warnings related to ICC profile
        with warnings.catch_warnings():
            warnings.filterwarnings("ignore", category=UserWarning, message=".*iCCP.*")
            img_arr = cv2.imread(img_path)

        # Check if the image was loaded successfully
        if img_arr is None:
            print(f"Failed to load image: {img_path}")
            os.remove(img_path)
            continue
```

Figure 10: Typical Preprocessing Code

3.2.3. **TRAINING**

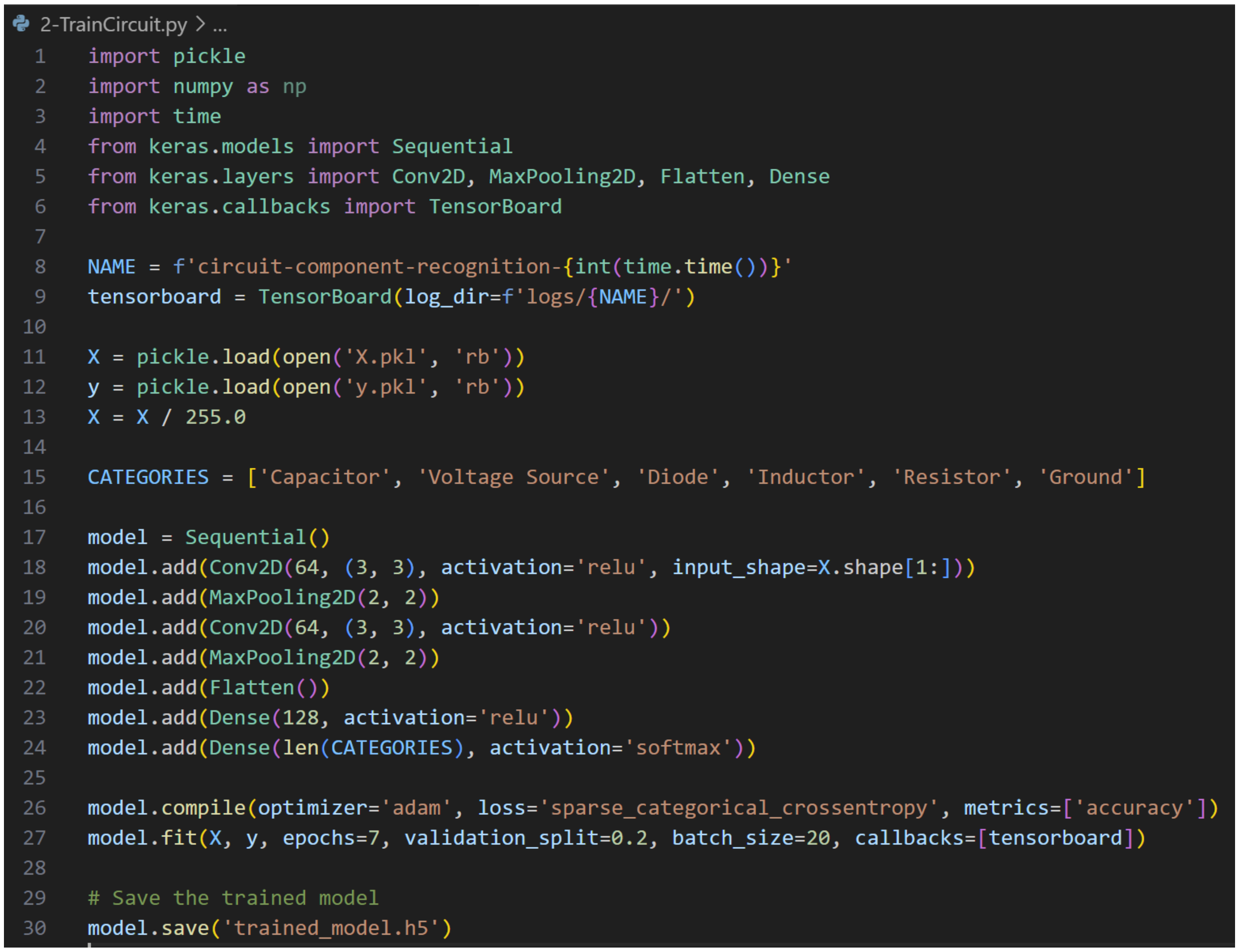
The script provided delineates the construction and training of a convolutional neural network (CNN) using Keras, a high-level neural networks API running on top of TensorFlow, which is particularly well-suited for deep learning tasks such as image recognition. In this instance, the network is tailored to recognize circuit components, a task that requires the ability to discern intricate patterns and features from visual input. The model is initialized as a `Sequential` object, which implies a linear stack of layers. It's designed with convolutional layers (`Conv2D`) that perform the convolution operation, extracting features by sliding over the input image matrix. This is followed by max pooling (`MaxPooling2D`) to reduce the spatial dimensions of the output volume, thus decreasing the number of parameters and computation in the network, and helping to prevent overfitting.

The network then flattens the matrix to a vector and passes it through fully connected (`Dense`) layers that perform classification based on the features extracted by the convolutional layers. The final layer uses the softmax activation function to output a probability distribution over the predefined categories of circuit components. The model employs the Adam optimizer for training, which is an adaptive learning rate method, and uses sparse categorical cross-entropy as the loss function, suitable for classification problems with multiple classes.

The training process is where the model learns to classify circuit components by adjusting its weights through backpropagation, based on the error it makes in predictions. The model is trained over seven epochs, which denotes the number of times the entire dataset is passed forward and backward through the neural network. The validation split of 0.2 indicates that 20% of the data is held back from training and used to evaluate the model's performance. This helps in gauging the model's ability to generalize to unseen data and prevents overfitting. The batch size of 20 determines the number of samples that will be propagated through the network before the optimizer updates the model parameters. Smaller batch sizes often lead to a decrease in the generalization error. Full python script can be seen below in **Figure 11.**

Upon the completion of training, the model is saved as an H5 file, which is a data format designed to store and organize large amounts of data. It is a convenient way of persisting models in Keras, as it contains the architecture of the model, the weights, and the training configuration, including the optimizer, allowing for the exact same model to be reloaded later.

Adjusting parameters like the number of epochs, batch size, and validation split can significantly affect the model's performance. Increasing the number of epochs may lead to better training results but also increases the risk of overfitting if the model learns patterns that are too specific to the training data. A larger batch size can lead to faster training but may also cause the learning process to settle into a less optimal convergence point, hence affecting the model's ability to generalize. Therefore, finding the right balance in these parameters is crucial for building an effective model. The TensorBoard callback offers an in-depth view of the training process, allowing for monitoring of the model's performance metrics over time, which can inform decisions about these trade-offs.


2-TrainCircuit.py > ...

```python
import pickle
import numpy as np
import time
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.callbacks import TensorBoard

NAME = f'circuit-component-recognition-{int(time.time())}'
tensorboard = TensorBoard(log_dir=f'logs/{NAME}/')

X = pickle.load(open('X.pkl', 'rb'))
y = pickle.load(open('y.pkl', 'rb'))
X = X / 255.0

CATEGORIES = ['Capacitor', 'Voltage Source', 'Diode', 'Inductor', 'Resistor', 'Ground']

model = Sequential()
model.add(Conv2D(64, (3, 3), activation='relu', input_shape=X.shape[1:]))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(len(CATEGORIES), activation='softmax'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X, y, epochs=7, validation_split=0.2, batch_size=20, callbacks=[tensorboard])

# Save the trained model
model.save('trained_model.h5')
```

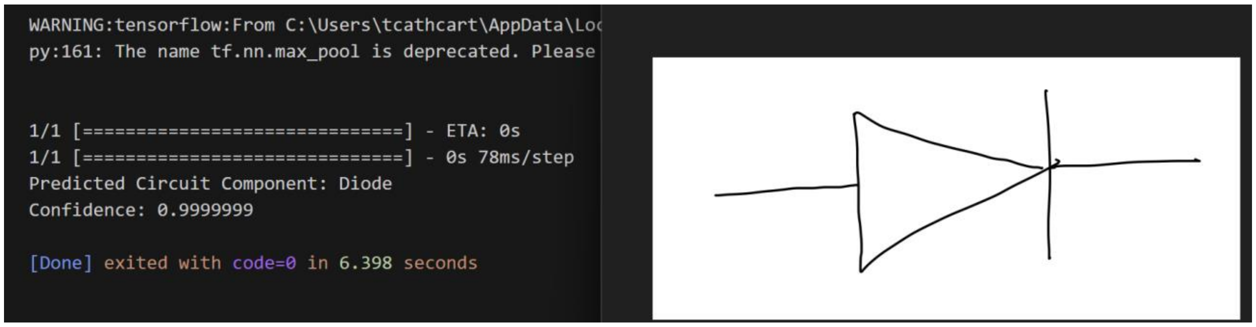
Figure 11: Train & Save Model Script

3.2.4. **TESTING**

After the meticulous training of a convolutional neural network model, the transition to practical application begins. The provided script embodies the next logical step, operationalizing the trained model to classify circuit components. This process starts when an individual sketches a circuit component on an iPad. Once it is completed, the image is saved to a directory that the script can access.

The script then engages in a straightforward routine. It first establishes the current working directory to ensure it can accurately locate the newly saved image. With the image path constructed and confirmed, the script utilizes OpenCV, a powerful library for image processing, to load the image. Prior to feeding the image into the model, the script preprocesses it to match the input specifications the model expects, based on the training regimen it underwent.

This includes resizing the image to a uniform 100x100 pixel size and converting it from BGR (Blue, Green, Red), which is OpenCV's default color space, to RGB, which is typically used in model training. An output with a guess from the model can be seen within **Figure 12.** By displaying the predicted label and confidence score, the script completes the loop from creation to recognition, demonstrating the real-world utility of machine learning. The artist's digital sketch, once just a collection of pixels, gains meaning through the model's eyes as it becomes identified as a specific circuit component. This powerful capability encapsulates the essence of machine learning—transforming raw data into understanding.



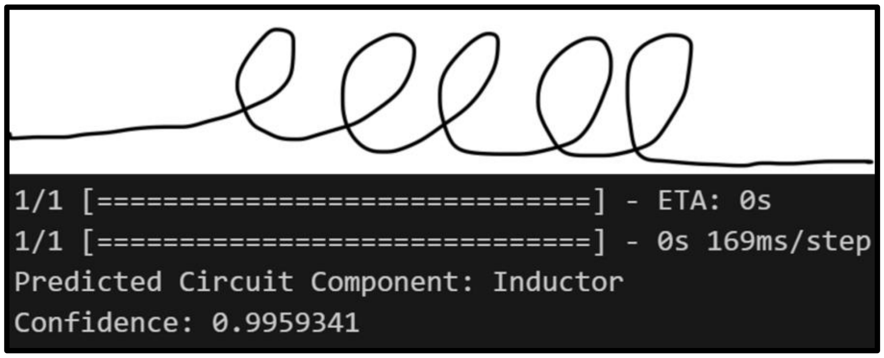
```
WARNING:tensorflow:From C:\Users\tcathcart\AppData\Loc
py:161: The name tf.nn.max_pool is deprecated. Please


1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 78ms/step
Predicted Circuit Component: Diode
Confidence: 0.9999999

[Done] exited with code=0 in 6.398 seconds
```

Figure 12: Testing Model Accuracy

3.3. EXPANDING MODELS

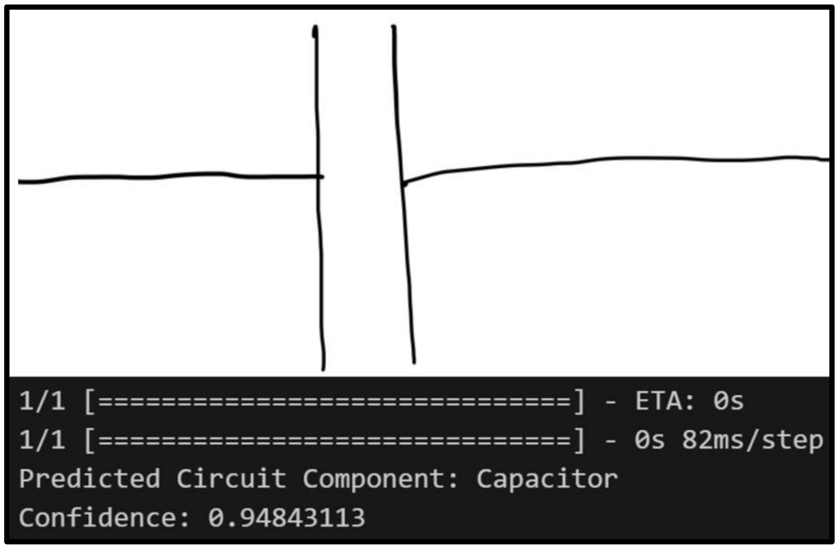
3.3.1. FURTHER TESTING

In our project update, we outlined how we refined our methods from the initial phase. Using our previously established image downloading techniques, we gathered more samples and subjected them to a series of enhanced augmentation processes. We then proceeded to train a more sophisticated model, applying these refined methods to achieve superior accuracy. The final model, built upon our deepened understanding and expanded dataset, demonstrated its proficiency by identifying individual circuit components in sketches with remarkable precision. Our collective efforts culminated in a system that not only recognizes but also understands the intricacies of circuit diagrams, as evidenced by the high confidence scores in our testing phase.This newly trained models test results can be observed in **Figures 13-16** for various single component tests.



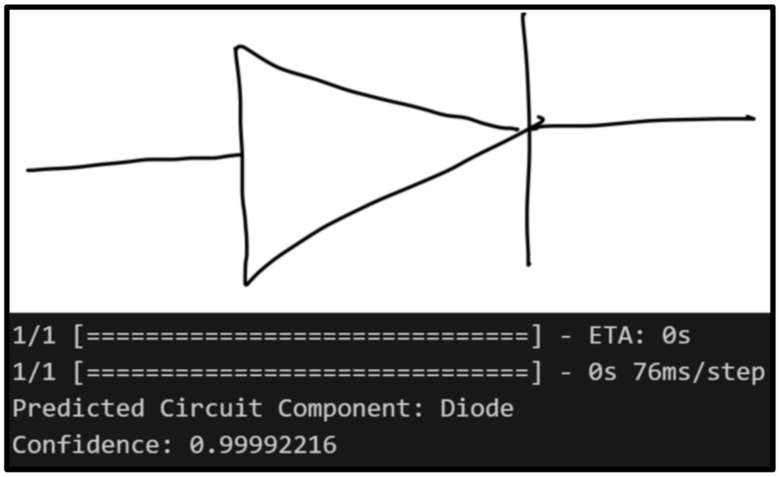
```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 169ms/step
Predicted Circuit Component: Inductor
Confidence: 0.9959341
```

Figure 13: Inductor Guess



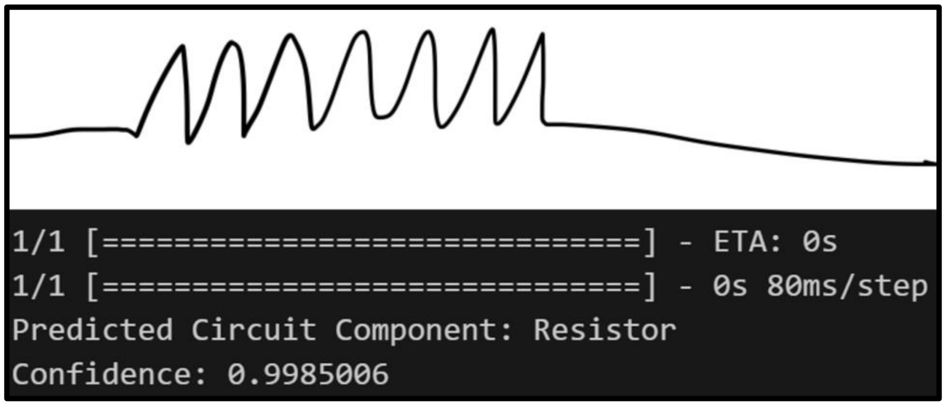
```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 82ms/step
Predicted Circuit Component: Capacitor
Confidence: 0.94843113
```

Figure 14: Capacitor Guess



```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 76ms/step
Predicted Circuit Component: Diode
Confidence: 0.99992216
```

Figure 15: Diode Guess



```
1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 80ms/step
Predicted Circuit Component: Resistor
Confidence: 0.9985006
```

Figure 16: Resistor Guess

3.3.2. **WINDOWING CIRCUIT**

In our project, we've advanced our approach to processing full circuit diagrams for component detection. We employ a windowing technique on the circuit input to allow our model to make educated guesses about the presence of different components. Due to variable component scales, we cannot rely on a fixed window size; instead, we dynamically adjust the window's dimensions. This method ensures that the entire component is likely to be within the window for the model to make an accurate prediction. As demonstrated in **Figures 17 and 18**, varying the window sizes is crucial. However, it's not a failsafe method for precise detection in every instance, and further refinements are required to enhance the model's accuracy consistently.

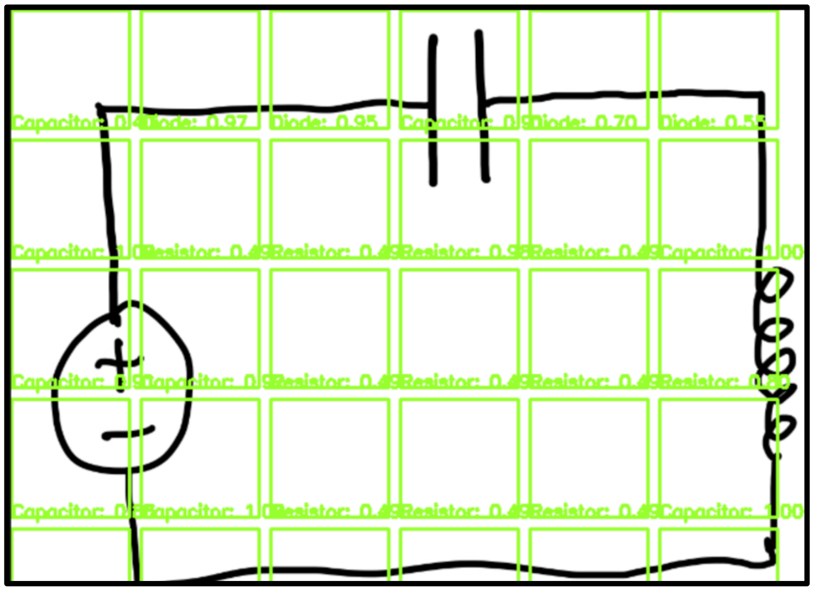


Figure 17: Smaller Windows

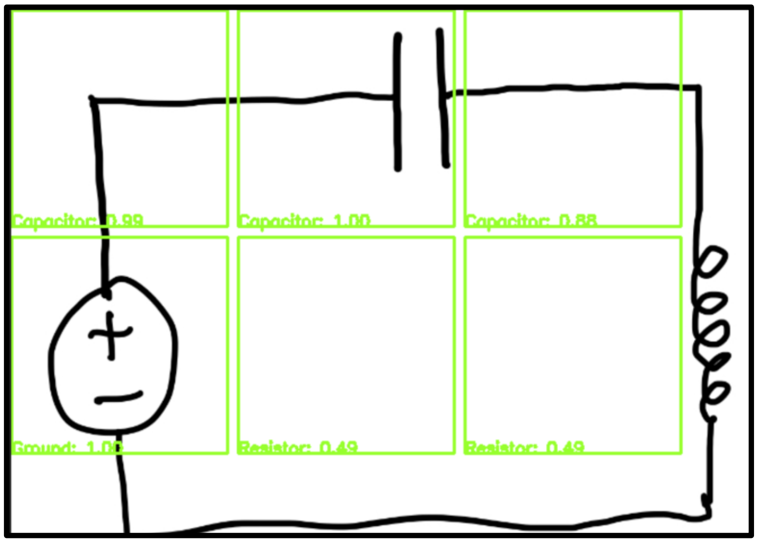


Figure 18: Larger Windows

3.3.3. **REFINING**

In our latest phase, we introduced a variable window sizing approach in our model to analyze entire circuit diagrams. This technique dynamically adjusts the window size during the scanning process, allowing the model to consider various scales of circuit components. The system then computes the average confidence across different window sizes for each detected component. By focusing only on the predictions with the highest confidence scores, we effectively pinpoint the most likely components within the circuit. This strategy is visually represented by green bounding boxes in the output image, which signify our model's most confident guesses, improving the accuracy of our component recognition system. The final output for a Low pass filter can be observed within **Figure 19,** and the accompanying code can be addressed in **Figure 20.**

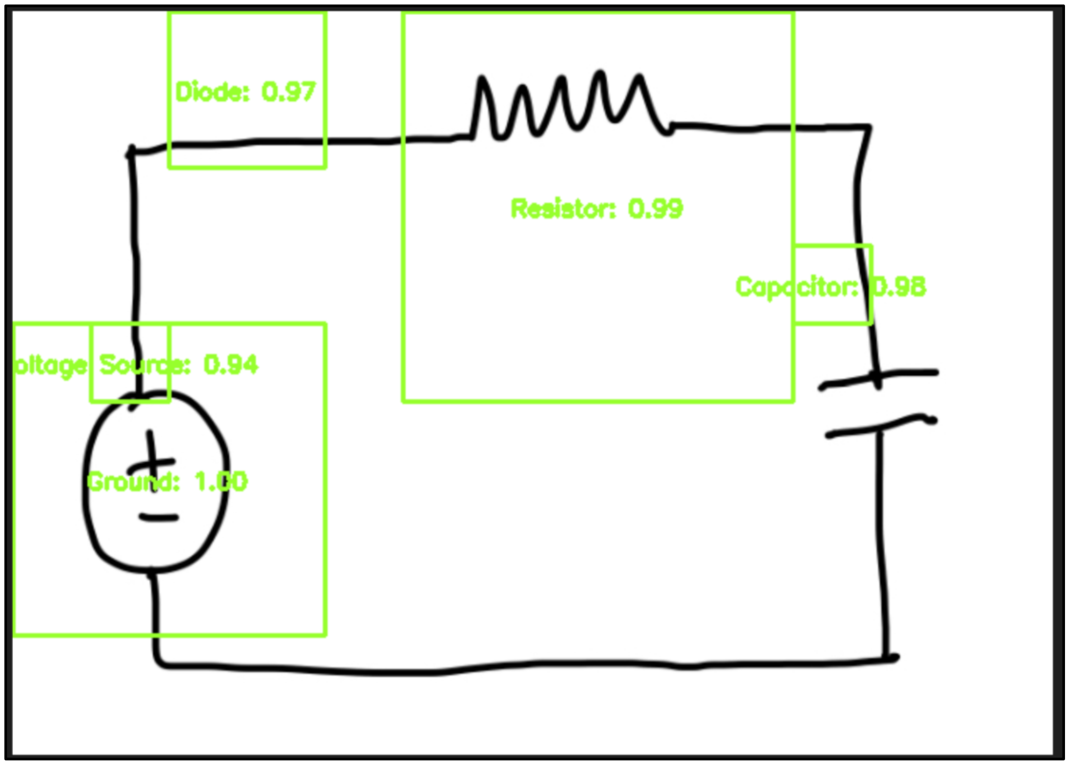
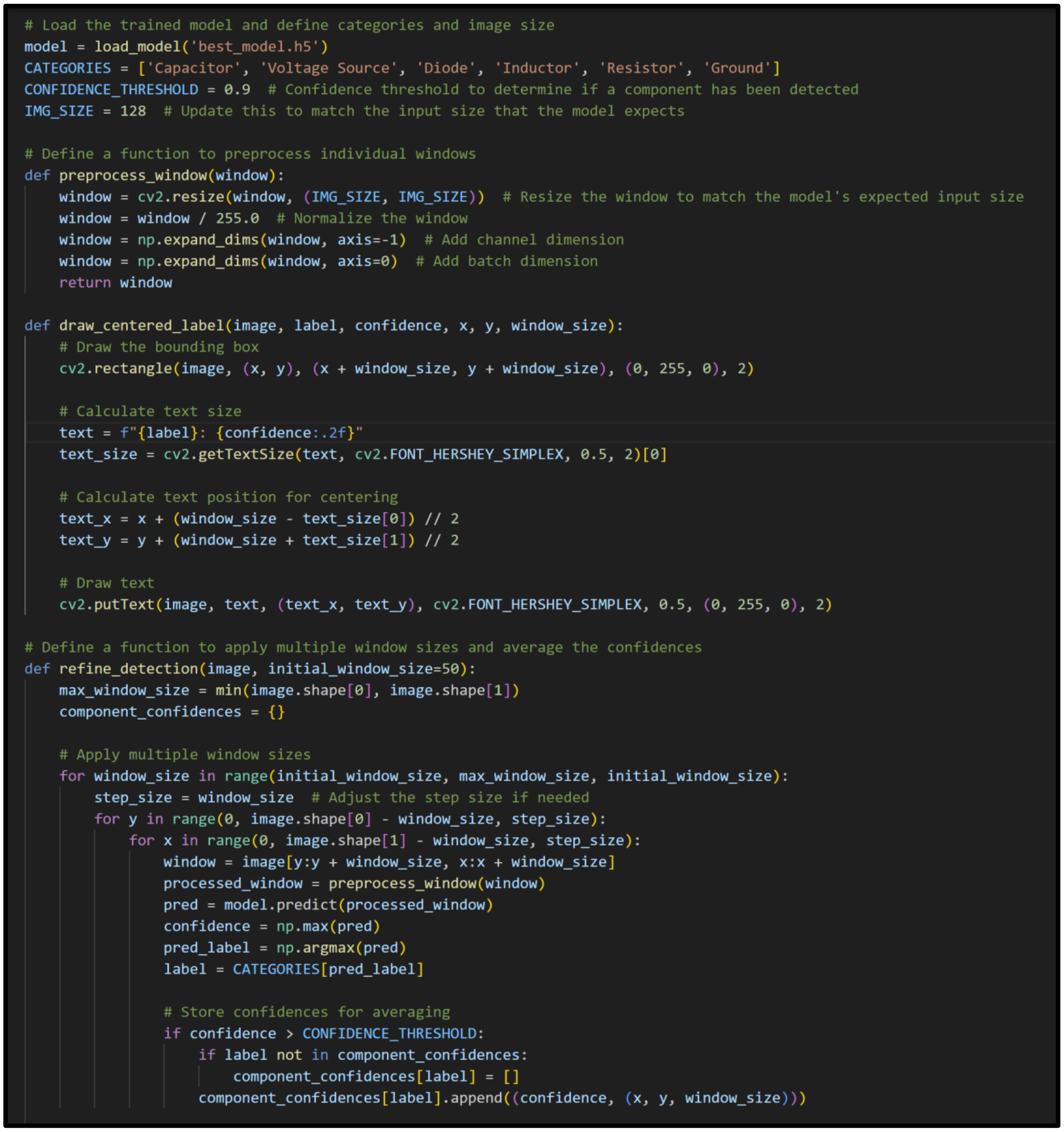


Figure 19: Final Windowing and Segmenting



```python
# Load the trained model and define categories and image size
model = load_model('best_model.h5')
CATEGORIES = ['Capacitor', 'Voltage Source', 'Diode', 'Inductor', 'Resistor', 'Ground']
CONFIDENCE_THRESHOLD = 0.9  # Confidence threshold to determine if a component has been detected
IMG_SIZE = 128  # Update this to match the input size that the model expects

# Define a function to preprocess individual windows
def preprocess_window(window):
    window = cv2.resize(window, (IMG_SIZE, IMG_SIZE))  # Resize the window to match the model's expected input size
    window = window / 255.0  # Normalize the window
    window = np.expand_dims(window, axis=-1)  # Add channel dimension
    window = np.expand_dims(window, axis=0)  # Add batch dimension
    return window

def draw_centered_label(image, label, confidence, x, y, window_size):
    # Draw the bounding box
    cv2.rectangle(image, (x, y), (x + window_size, y + window_size), (0, 255, 0), 2)

    # Calculate text size
    text = f"{label}: {confidence:.2f}"
    text_size = cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX, 0.5, 2)[0]

    # Calculate text position for centering
    text_x = x + (window_size - text_size[0]) // 2
    text_y = y + (window_size + text_size[1]) // 2

    # Draw text
    cv2.putText(image, text, (text_x, text_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# Define a function to apply multiple window sizes and average the confidences
def refine_detection(image, initial_window_size=50):
    max_window_size = min(image.shape[0], image.shape[1])
    component_confidences = {}

    # Apply multiple window sizes
    for window_size in range(initial_window_size, max_window_size, initial_window_size):
        step_size = window_size  # Adjust the step size if needed
        for y in range(0, image.shape[0] - window_size, step_size):
            for x in range(0, image.shape[1] - window_size, step_size):
                window = image[y:y + window_size, x:x + window_size]
                processed_window = preprocess_window(window)
                pred = model.predict(processed_window)
                confidence = np.max(pred)
                pred_label = np.argmax(pred)
                label = CATEGORIES[pred_label]

                # Store confidences for averaging
                if confidence > CONFIDENCE_THRESHOLD:
                    if label not in component_confidences:
                        component_confidences[label] = []
                    component_confidences[label].append((confidence, (x, y, window_size)))
```

Figure 20: Final Windowing and Segmenting Code

4. ACKNOWLEDGEMENTS

In the journey of bringing the Machine Learning Circuit Component Recognition project to fruition, the guidance and expertise of Dr. Dua, a distinguished member of the Electrical Engineering Cooperative Program at Missouri S&T, have been invaluable. His profound knowledge in electrical engineering and digital image processing illuminated paths through the most challenging aspects of this research. At moments when progress seemed to falter, Dr. Dua's insights and recommendations provided the necessary direction and clarity to advance.

Our regular monthly meetings were not merely checkpoints; they were opportunities for learning and exploration beyond the immediate scope of the project. Dr. Dua's commitment to sharing his deep understanding of electrical engineering, combined with his expertise in digital image processing, played a pivotal role in overcoming the technical hurdles encountered during the project. His contributions extended far beyond mere advisory; they were foundational to the project's success, offering a blend of mentorship, knowledge, and support that was critical at every stage.

Moreover, Dr. Dua's willingness to engage deeply with the project, offering insights not only on the specific challenges faced but also broadening my understanding of electrical engineering as a whole, has enriched my educational journey immeasurably. This project, under Dr. Dua's mentorship, has been a profound learning experience, shedding light on the complexities of machine learning applications in electrical engineering and providing a solid foundation for future explorations in the field.

I extend my deepest gratitude to Dr. Dua for his unwavering support, guidance, and for sharing his wealth of knowledge. The completion of this project would not have been possible without his invaluable contributions.

5. REFERENCES

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press. https://www.deeplearningbook.org/

2. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778). https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

3. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. https://www.nature.com/articles/nature14539

4. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

5. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556. https://arxiv.org/abs/1409.1556

6. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9). https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_CVPR_2015_paper.html

7. Chollet, F. (2017). Deep Learning with Python. Manning Publications. https://www.manning.com/books/deep-learning-with-python

8. Gonzalez, R. C., & Woods, R. E. (2018). Digital image processing (4th ed.). Pearson. https://www.pearson.com/store/p/digital-image-processing/P100000648488

9. Bishop, C. M. (2006). Pattern recognition and machine learning. Springer. https://www.springer.com/gp/book/9780387310732