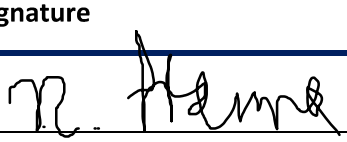
Toronto Metropolitan University

Department of Aerospace Engineering
Faculty of Engineering & Architectural Science

Semester (Term, Year)	Fall 2024
Course Code	AER850
Course Section	02
Course Title	Intro to Machine Learning
Course Instructor	Dr. Reza Faieghi
Submission	Project
Submission No.	1
Submission Due Date	Dec 16th, 2024
Title	Project 3: Object Masking & YOLO v8
Submission Date	Dec 16th, 2024

Submission by (Name):	**Student ID (XXXX1234)**	**Signature**
Nathanael Hanna	XXXXX0646	

By signing the above you attest that you have contributed to this submission and confirm that all work you contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, and "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Academic Integrity Policy 60, which can be found at www.torontomu.ca/senate/policies/

Table of Contents

List of Figures

1 Introduction

The objective of this report is to detail the process by which a) Python's CV2 edge & contour detection were used to detect, isolate, and extract a motherboard from a photo; and b) Yolov8 Nano object detection model was created in Python, with the objective of visually classifying components on a Printed Circuit Board (PCB) into several categories The data given consisted of 544 training images, 105 validation images, and 22 test images. The model weights were determined using these images, and then a final set of evaluation images (three total; two Arduinos and one Raspberry Pi) were run through the model to gauge the effectiveness of the weights for daily use.

The training of a YOLO model, even a Nano model, requires a significant amount of computation and is infeasible for even most modern CPUs. As the author of this code does not currently have access to a CUDA-enabled GPU, code for this program was created in Google Colab to leverage the processing power of Nvidia T4 Quadros during model training. Google Colab allows for the separation of code into cells, which function much like sections in a typical IDE and allow for individual blocks of code to be run. This function was used to split up Parts I, II, and III, as well as any supplementary code that needed to be run (installs, debugging, etc.).

2 Theory

For Part I: Object Masking, the basic premise is the use of OpenCV to detect a motherboard in an image and extract it into an image of its own, without the background. This is accomplished by a series of algorithms that either detect the corners of objects through rapid changes in color between pixels, or through algorithms that detect the contour of an object by spotting lines of consistent colors adjacent to other colors [4].

The potential benefit of this process is obvious when considered in relation to Parts II & III of this project: by isolating a target object and removing the background, object detection models like YOLO can be more accurately trained, and once deployed, the addition of object masking into the preprocessing stage can increase overall accuracy, not to mention decreasing computational load.

For Part II: YOLOv8 Nano Training, the process involved using a prebuilt model (The YOLOv8 Nano model for which the section is named) and training that model on an image set similar to the intended use case. As the model is pretrained, no adjustments to the structure of the model were made, but the dataset allows for customization by way of training the model on those specific images to adjust its weights in a way that are best-suited for the task. Like training any model, the training images are manually annotated and allow the model to "learn" the components being detected, after which it attempts to replicate the desired results on the validation images. Other than the images, the only modifications that can be made to a YOLO model are tuning parameters used during training. These consist, among others, of epochs, image size (imgsz), batch size, patience, and several file sorting/debugging tools. These will be discussed in further detail in section 3.2.

The use-case for this procedure is almost unending, as it allows for the (relatively) rapid training of a neural network that can detect objects with speed and accuracy far exceeding that of a single human, or even an entire assembly line. Aside from detecting defects in PCBs and flagging them for review, other uses of object detection include automatic photo editing, facial recognition, or—as discussed in class—self-driving vehicles. Professional or industrial machine learning models are probably custom-built, but the fact that the YOLOv8 model is as easy and accessible as it is to individual users allows for a great variety of projects or scenarios.

3 Method

This section details the step-by-step process by which The motherboard mask was created, and how the YOLOv8 model was trained. The results of the image mask are located in section 4.1, while the YOLOv8 results are located in section 4.2.

3.1 Part I: Object Masking

As discussed by the professor in class, two main masking techniques were tested: edge detection and contour detection. Each technique comes with its own strengths and weaknesses, but ultimately contour detection was chosen for its simplicity and ease of use. It was originally expected that edge detection would be a superior choice given the prevalence of right angles and overall “blockiness” of the motherboard. However, counterintuitively, in testing the edge detection methods seemed to favor capturing the background of the photo over the motherboard itself, which made for a difficult and unhelpful tuning process. On the other hand, contour detection turned out to be useful for an unexpected reason: the motherboard, with all the traces and sockets and multicolored connectors, produced some surprisingly detailed contours. This allowed for the use of the [largest_contour] tag to isolate the motherboard, or at least a significant portion of it. However, this came at a cost: the contour selected contained several cutouts, akin to a fjord on a map, that obscured parts of the motherboard. However, the four outer corners of the motherboard were captured with adequate accuracy. Upon further reading within the OpenCV documentation, a method was discovered that would allow the removal of the cutouts while still maintaining the shape of the board: cv2.convexHull(largest_contour) on line 31 created a bounding box around the contour that stayed as close to the original mask as possible but without a single concave corner [4]. In effect, it created a rough polygon, and one that is not limited to 90 degree angles such as cv2.boundingRect or cv2.minAreaRect, which captured too much area outside of the motherboard on account of it being at an angle in the photo. Once a convex hull was used to create the mask, it was simply a matter of applying the mask to the motherboard image and outputting a new photo.

3.2 Part II: YOLOv8 Nano Training

This section comprises the bulk of the report in terms of time required, which is interesting considering it only required ~20 lines to code. However, as expected the pretrained model still requires a significant amount of time to adapt to the dataset provided, leading to some code runtimes that approached four hours. Because of this, the professor's suggestion to use a YOLOv8 Nano model was gladly accepted, as the larger YOLOv8 models may have taken longer yet to fully train, at which point Google Colab would have terminated the runtime multiple times over.

The first step of the process was to import the YOLO model, which required running [pip install ultralytics] (run in a separate code cell, for organization), then importing the model into a data file. Once done, a directory was established (named [checkpoint_dir] on account of its primary purpose) and the model was called for training. Within [model.train], eight separate variables were set:

1. [data]: points to the directory where the .yaml file is located, for training and validation.
2. [epochs]: sets the number of total runs the model uses to train itself (strongly recommended to set < 200 by the project description, likely to prevent overfitting and wasting Colab resources). 164 was chosen for this project.
3. [imgsz]: image size, in pixels. Allows downscaling to speed training. Resolutions < 900 were not recommended due to the detail required in component detection. 1280 was chosen for this model to allow for adequate detail without wasting computational resources on the native resolutions, some of which are significantly larger.
4. [batch]: batch size allows you to determine how often the model updates its weights within a single epoch, by specifying how many images it trains on at once. Several attempts were made during training to use a batch size greater than 4, for better speed and model generalization. Unfortunately, the T4 Quadros used in Google Colabs have 16GB of vRAM, which was maxed out by batch sizes of 16, 12, 8, 6, and 5, in all cases leading to rapid crashes. Thus, a batch size of 4 was simply what was allowable by hardware constraints.
5. [patience]: allows you to set an early stopping point for the model, so that if the metrics between epochs don't improve for a certain number of iterations, the model weights are locked in to prevent overfitting. A patience of 8 was set for this model.

6. [name]: simply specifies the name of the model and the directory it is stored in.
7. [project]: this variable instructs the model to save the weights in a specific location. Normally, the model saves the last and best weights within Google Colab. This caused problems at one point during testing when the Colabs runtime ran out and the model weights were wiped after over 3 hours of training. By specifying the project directory and pointing it to a Google Drive, the weights were saved and training could be resumed in the event of failure.
8. [save_period]: this instructs the model to save weights at a certain interval regardless of last or best weights. This has applications in research and debugging, but was set in this particular instance as an extra level of redundancy after the aforementioned problems encountered. This model was instructed to save weights every 32 epochs.

One note to be made regarding the code is the presence of several commented out lines that are mostly identical to adjacent lines, with the exception of pointing to different file directories. As previously alluded to, this was the result of Google Colabs usage limits forcing the use of multiple accounts to train the model in a timely manner.

4 Results & Discussion

This section outlines the results of both the object masking process (4.1) and the YOLO v8 Nano training process (4.2).

4.1 Part I: Object Masking (results)

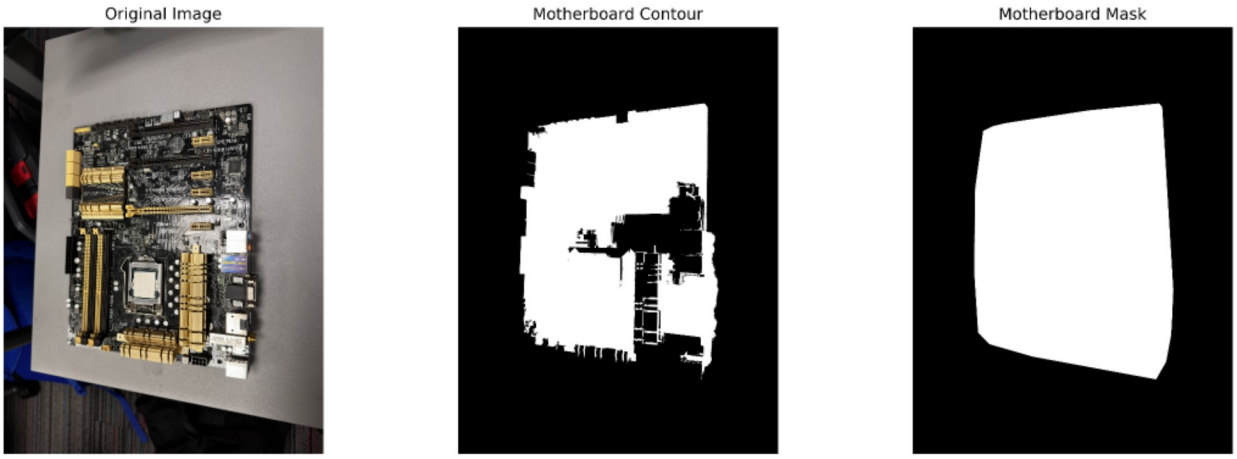


Figure 4.1: Original image, contour detection mask, and hull mask.

Figure 4.2: Isolated Motherboard.

As shown above, the motherboard contour seemingly traced a path around most of the motherboard, but also cut out sections where the gold plating or light reflections raised the average light value beyond the threshold. Testing was done to try to mitigate this, but changing the threshold too drastically resulted in irrelevant contours being selected from the background image, which is why the hull contour was ultimately selected for use in isolating the motherboard from the original contour. The final isolated motherboard (Figure 4.2) is a fairly tight contour, but it does also include some extra shadow along the close side of the motherboard, and it cuts off a small amount of the gold plating on the far side. Some imperfection is expected: the gold coating is a much brighter color value than the rest of the black PCB, which causes difficulty for contour detection that factors brightness into the equation. This would be even worse for detection models based on brightness alone: early testing was conducted using a black & white image, which yielded extremely poor results. The motherboard image in the project description also included some extra shadow and over-extracted around the gold sections, suggesting that some level of imperfection is unavoidable. In this case, further improvement could potentially be gained by subjecting the isolated motherboard to another round of masking that would allow for concave corners without confusion from extraneous background information. By adding extra layers of trimming and masking before arriving at the final mask, the system may have an easier time discarding extraneous information without going overboard.

4.2 Part III: YOLOv8 Nano Evaluation

The results of the YOLOv8 Nano training were by no means perfect, but they do highlight the excellent performance of even the smallest of v8 models, showing surprising accuracy even after a relatively short training period with hardware limitations forcing some dubious training parameters.

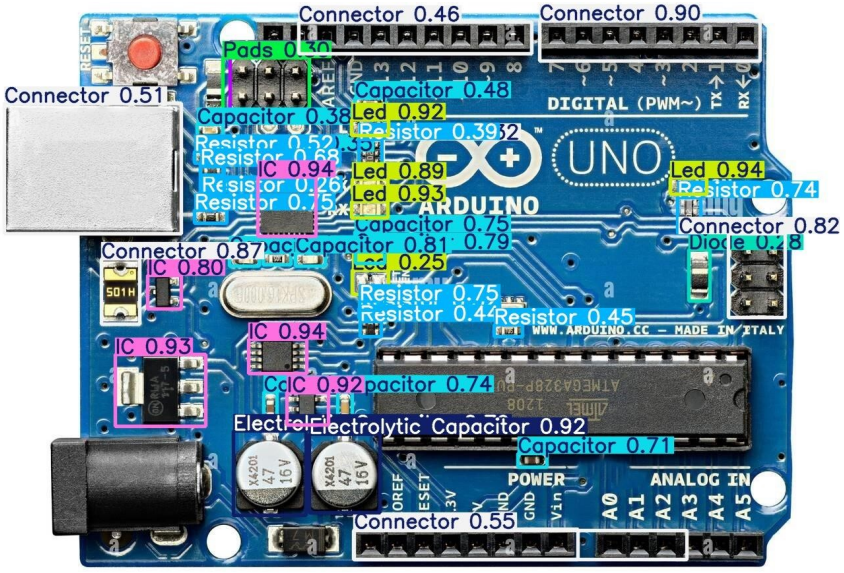


Figure 4.3: Arduino UNO results.

The Arduino UNO was arguably the poorest result of the lot, with three very obvious missing components: the upper-left button, the connector in the lower left, and the capacitor in the left-center of the PCB. Additionally, many of the confidence scores were lower than they should realistically have been, suggesting that a lower resolution photo or one taken in worse conditions may have yielded much poorer results. That said, overall most components were spotted and labeled correctly, and there were no false positives.

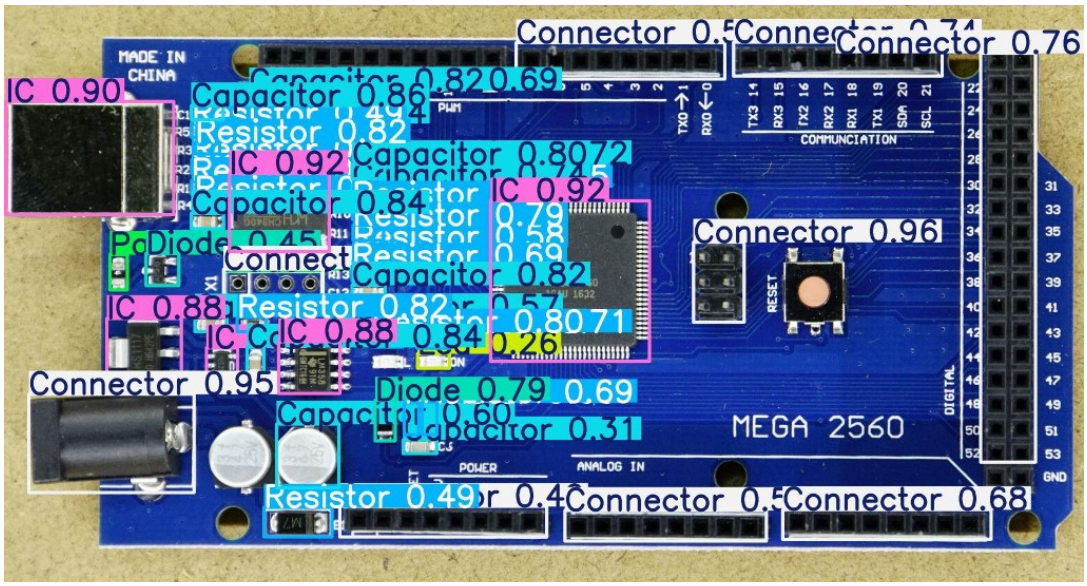


Figure 4.4: Arduino MEGA results.

The results of the Arduino MEGA are similar to that of the UNO, with a few unfortunate omissions in the left corners/center of the PCB. That said, there are no obvious false positives, and even one of the "missed" connectors in the upper-left corner was still partially detected.

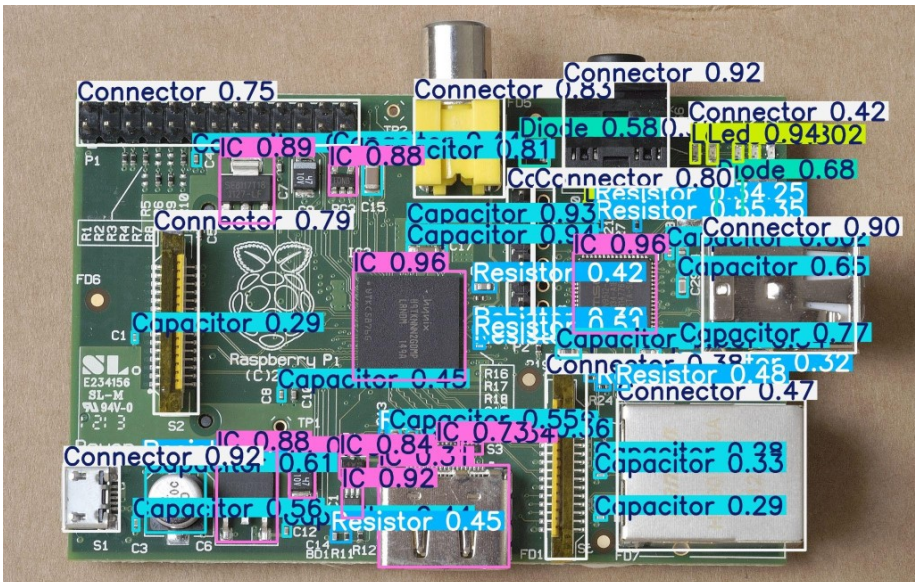


Figure 4.5: Raspberry Pi results.

The Raspberry Pi is seemingly the most promising result for this model, with no obvious errors of any kind and decently high confidence scores across the board, excepting capacitors. That said, even those capacitors look to be correctly labeled, suggesting that although the model

is not very confident in itself, it is generally correct. Like the previous two photos, there are few to no false positives. One IC is missing in the top-middle-left, but that seems to be the only missed component. Overall, an excellent result.

The three results above, particularly in their missed components, low confidence scores, and lack of false positives, suggest that the model developed some very cautious weighting parameters, preferring to miss a component than incorrectly label something that is not there. This may be in part due to the Nano model being smaller and less sophisticated than other v8 models, which may offer a bit more “neural horsepower” to detect and classify ambiguous components. The same may also apply to a model trained with more epochs.

In the future, the accuracy of these results could be further improved through several means: using a larger v8 model, using a GPU with more vRAM and increasing the most tuning parameters (batch size, epoch count, patience, and image size, to name a few), or manually creating a larger training set. Of these, using a better GPU and increasing tuning parameters would likely be the best option, since larger batch sizes would increase the training speed of the model and allow for more epochs, and a higher resolution allows for more detail to be retained in the training data.

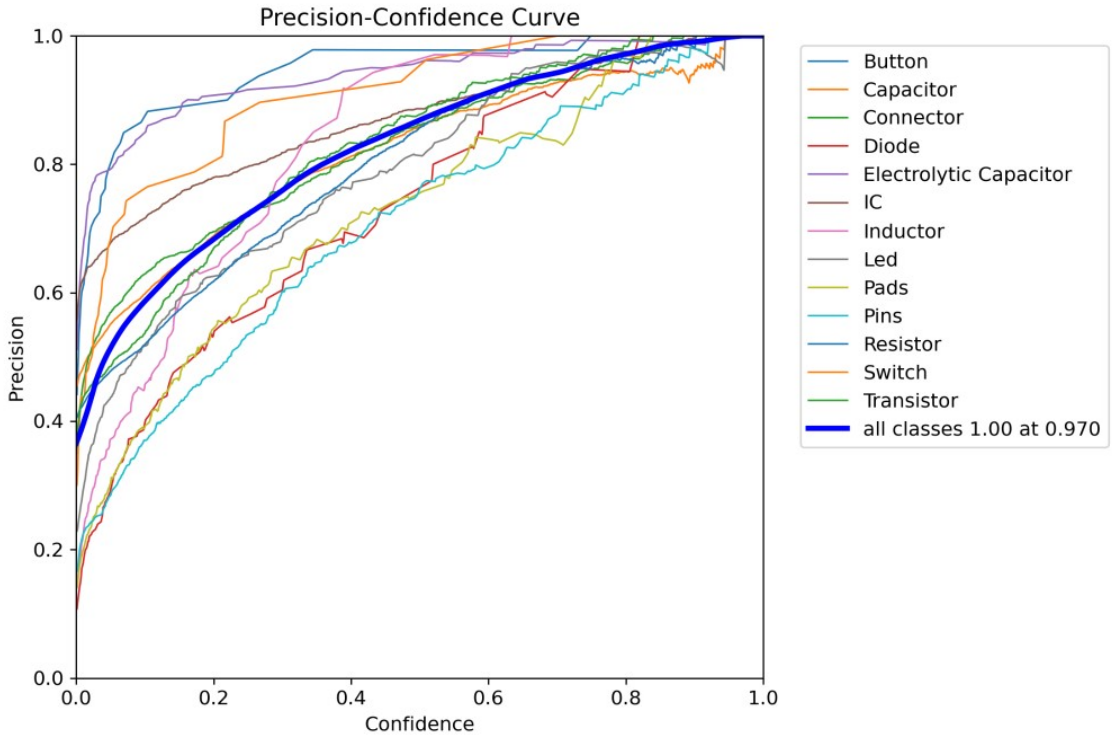


Figure 4.6: Precision-Confidence curve.

The precision-confidence curve is, in a nutshell, a measure of how right the model thinks it is vs. how right it actually is. High confidence means the model is certain of a component, and high precision means the model is usually classifying those components correctly. For example, on the graph above, when the model just barely thinks a component is a switch, for example (confidence approaching 0.0), it was still correct nearly 50% of the time. On the other side of the graph, any component predicted with a > 95% confidence was almost certainly correct (precision approaching 1). This model, already suspected of being somewhat cautious in its prediction, reinforces that belief with even the most dubious predictions starting at 40% accurate and rapidly scaling up from there.

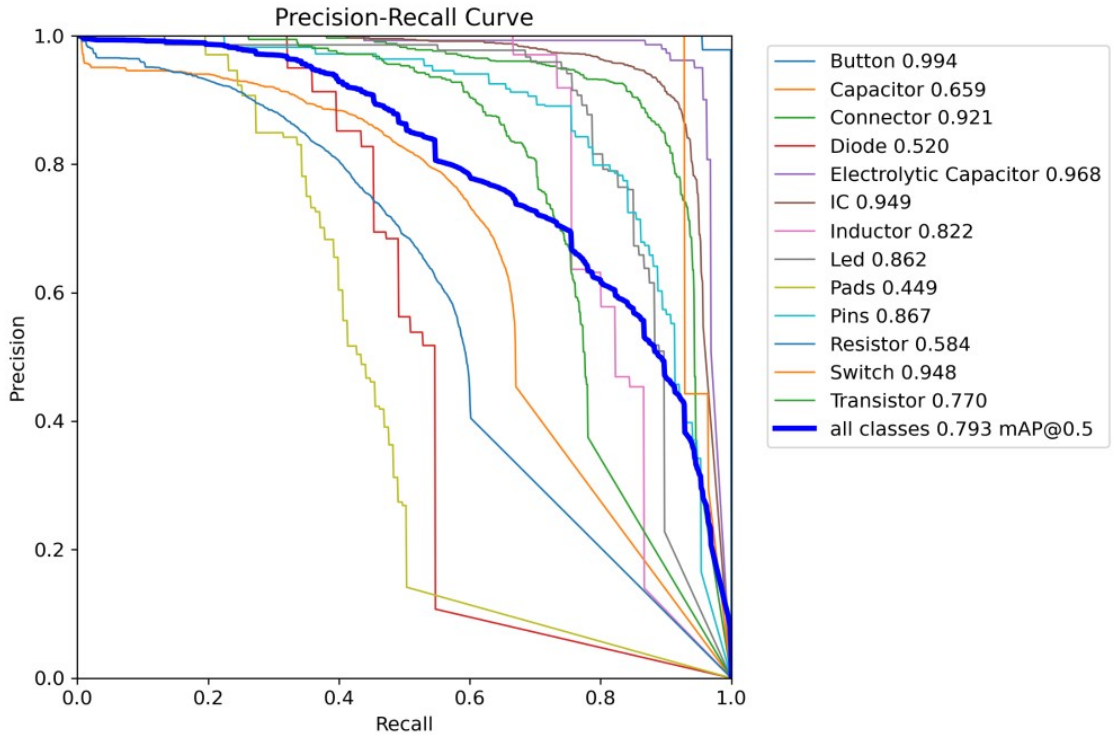


Figure 4.7: Precision-Recall curve.

The precision-recall curve is a measure of how many components it missed (recall: correct positives / all real positives) vs. how many times the model "overshot" and made bad predictions (precision: correct positives / all predicted positives) [3]. A precision approaching 0 and a recall approaching 1 means that the model threw a label on *everything*, even random noise or other components. Likewise, a precision approaching 1 and a recall approaching 0 means that the model was far too cautious and barely predicted anything at all. Understandably, these metrics should both be as close as 1.0 as possible, which would mean the model had, respectively, no false negatives and no false positives.

The 'averages' line on the graph reaches closest to the top right corner at roughly (0.75, 0.70), which indicates a recall of 75% and a precision of 70%. This is a very interesting result, because it contradicts the earlier assumption that the model was "cautious": a cautious model would have a high precision and a low recall, not vice versa. That said, the massive variability between individual components shown here suggests that perhaps some weighting is at play, and some components appeared more than others to skew the results.

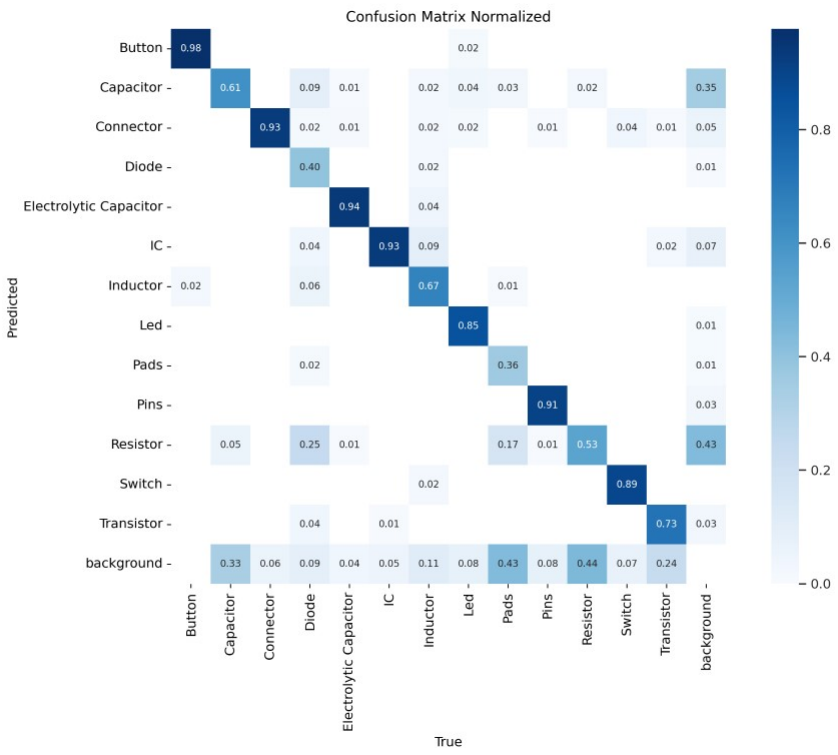


Figure 4.8: Normalized Confusion Matrix.

The confusion matrix is a graphical representation of the correct and incorrect predictions the model made during its validation. Results along the center diagonal are good: those components were predicted to be the same class they truly are. Anything else is an error, and if a significant number of errors appear in a certain section, that means the component in question is being notably over-or-underpredicted. A normalized confusion matrix, shown above, takes the confusion matrix and normalizes the values into decimals, allowing for a quick comparison that shows how each percentage of each component is being categorized without the extra noise generated by some components being more common than others.

From the above, the usual center diagonal confirms most of the predictions were made correctly, particularly buttons, with an almost perfect 98% accuracy. That said, some components fared quite poorly, with two components showing sub-50% accuracy and pads in particular only being identified 36% of the time. 43% of the pads shown were incorrectly dismissed as background noise, and judging by the line along the bottom of incorrect “background” predictions, the model once again appears to be overly cautious and hesitant to classify components it should have. Of interest as well are the clusters of background noise incorrectly

classified as capacitors and resistors. Neither of those errors were obviously present on the evaluation images, but if this model were being pushed into deployment, caution would be prudent in case the model continues to overpredict.

5 Conclusion

In this project, the objective was, firstly, to develop an object masking feature capable of detecting a motherboard and isolating it into a separate image. The next objective was to create a machine learning model trained to notice and classify components on a PCB.

In the first task, the motherboard was detected to an adequate degree of accuracy, with a few corners slightly shaved off due to lighting and coloration issues. This may be solved in future iterations through a more advanced object detection algorithm, more masking steps, or even just taking better photos.

In the second task, the model trained was able to accurately detect the most components, with the majority of errors in both the evaluation and validation datasets to be centered around missing components instead of over-classifying. This could be improved in the future through several different methods all focused on increasing model training, whether through increasing resolution, batch size, epochs, etc. to increase the overall confidence level of the model and allow it to make more predictions.

6 References

[1] abhishek1, “OpenCV tutorial in Python,” GeeksforGeeks, https://www.geeksforgeeks.org/opencv-python-tutorial/ (accessed Dec. 08, 2024).

[2] ChatGPT-4o, https://chatgpt.com/ (accessed Dec. 12, 2024).

[3] GeeksforGeeks, “Precision-recall curve: ML,” GeeksforGeeks, https://www.geeksforgeeks.org/precision-recall-curve-ml/ (accessed Dec. 16, 2024).

[4] “OpenCV modules,” OpenCV, https://docs.opencv.org/4.x/index.html (accessed Dec. 14, 2024).

[5] R. Faieghi, Project #3. Toronto, ON.

[6] Ultralytics, “Yolov8,” YOLOv8 - Ultralytics YOLO Docs, https://docs.ultralytics.com/models/yolov8/ (accessed Dec. 13, 2024).

My GitHub repository: https://github.com/Nathanael-e-h/NHanna_AER850_Project_3

NOTE: The references above are listed in alphabetical order instead of the traditional “order referenced” method. They are not necessarily cited in the paper (but some are); instead, they are the resources & documentation used to create the machine learning program which the report is based on.