EEL 5840 Mathematical Symbols Final Project Report

Atayliya Irving, Catalina Murray, Justin Rossiter, and Ceenu Shaji[1]
[1]Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA

For our final project, we implemented an algorithm for classification of handwritten mathematical symbols, using a dataset compiled from our class. We implemented YOLOv5 for classification of these handwritten symbols in addition to object detection of symbols for classifying multiple symbols in an image. We implemented code to generate a filesystem structure from the provided data for training and validation with corresponding bounding boxes generated through image preprocessing. We also implemented data augmentation with the intent for better training on spatial relationships between symbols in an image. Our model performance yielded 62 percent accuarcy on a dataset of images with multiple symbols, and yielded 96 percent accuracy on a single-symbol classification test set generated from our provided dataset compiled by the course.

***Index Terms*—Machine Learning, Transfer Learning, Object Detection**

I. Introduction

THE goal of this research was to train a dataset of mathematical symbols that would yield reliable test results and correctly recognize those symbols. The data set was collectively compiled by the students enrolled in Fundamentals of Machine Learning during the fall 2022 semester. After we fixed the data mislabeling, our biggest goal was to find the best training model. As we wanted our model to be competent for the extra credit, we decided YOLO would be the best training model.

A. YOLO

"You only look once" (YOLO) is an object recognition technique that relies on a grid to pinpoint the locations of images' contents. Everything inside the bounds of a single grid cell is the responsibility of that cell alone. The objective of multi-object identification is to locate all potential objects and the comparing region in a picture [2]. Targets are stated to be the items having the highest probability (above the predefined boundary) [4]. Single-stage and two-stage IDs can be used to identify the multi-object location [3]. The single-stage identification disregards the inquiry into the region's suggested location, which is the initial step of the two-stage identifier [2].

Deepening the neural network decreases the discovery accuracy when a spatial objective is highlighted poorly. YOLO calculations make full advantage of element guides by relating component maps of varying sizes to the image's bounding box [2].

B. Choosing the best YOLO

While we were certain to employ YOLO, each of its variations was superior than the others. We had to carefully consider which option best fits our project, through a thorough literature review on this topic. The original three YOLO variations were supplied separately in 2016, 2017, and 2018. However, in 2020, three key versions of YOLO were released: YOLOv4, YOLOv5, and YOLOv6 [2]. YOLOv5 was implemented by Glenn Jocher on 18 May 2020. Specifically, YOLOv5 has a loading record of 27 megabytes, while YOLO v4 (with Darknet design) has a loading record of 244 MB. YOLOv5 is approximately 90 percent more reserved than YOLO v4[2]. Since YOLOv5 is initially implemented in PyTorch, it benefits from the established PyTorch ecosystem in terms of support and structure. YOLOv5 is also known to be the best "small" object identification algorithm [2]. In addition, since YOLOv5 is a more well-known research framework, stressing it may be easy for the larger research community [2].

C. Approach

YOLO considers image detection to be a relapse problem, and hence it prioritizes a simple pipeline and high speed. YOLO's key idea is to use the big picture as the organization's contribution and then circle back to the context of the leaping box and the category to which the jumping box belongs in the aftermath [1]. Each YOLO bounding box has five expectancies and confidences that are comparable to the lattice unit at the limit's focal point, and each bounding box is anticipated by the characteristics of the entire image [2].

II. Implementation

Our implementation method can be divided into three main sections: preprocessing our data, training our model, and testing our model.

Our data included 9032 images of handwritten mathematical symbols compiled by the class at the start of the semester (refer to figure [1]). In our pre-processing stage of the project, we created an unknown class which included any image that was either blank or had a character not a part of one of our original classes of math symbols (class 0-9); we labeled these images as class 10. We found approximately 32 images which we labeled as the unknown class. The distribution of classes 0-9 are around 900 images per class after preprocessing; the label distribution is shown in figure [2].

We also resized the images into 320x320, this is because the method we used to train our model YOLOv5 requires images that are sizes of increments of 32; this preprocessing was performed with the Pillow library.

Math Symbol	Label	Integer Encoding
x	x	0
$\surd$	square root	1
$+$	plus sign	2
-	negative sign	3
$=$	equal	4
%	percent	5
∂	partial	6
$\prod$	product	7
π	pi	8
$\sum$	summation	9

Fig. 1. Handwritten symbols and corresponding labels for this project. In our implementation, we included a tenth class for symbols and detections that are none of the above

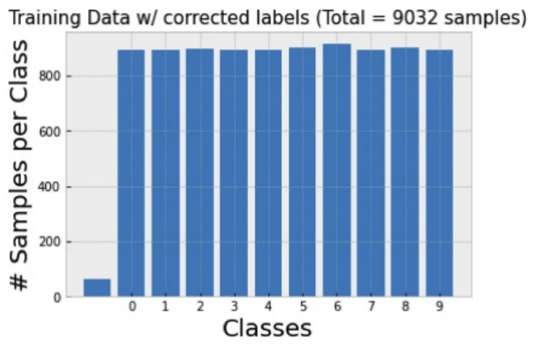

Fig. 2. Number of samples per class, the unlabeled class is the ”unkown class” which ended up giving it a label of 10

Next, we utilized a Jupyter Notebook, ”train.ipynb” to first create bounding boxes for each image to be used when training the model. train.ipynb also splits the data into training, validation and test sets, and creates a folder structure for the train and validation sets with the images and their corresponding boxes. The test dataset and its corresponding labels are saved to NumPy arrays to be loaded in test.ipynb for evaluation.

To create the bounding boxes we first invert the grayscale image, and then apply global thresholding to binarize the image. In implementation, we used the Otsu thresholding implementation within the library scikit-image. This uses a histogram to find the optimal threshold value by finding the maximum variance between two classes of pixels in an image. Any pixel that is below the threshold become a 0 and any value above the threshold becomes a 1. After applying the thresholding, we perform dilation and erosion on the image. Then find the left-most, right-most, top-most, and bottom-most column containing a nonzero pixel and we assume those points to be at the edge of the symbol. These points helps us to create a bounding box surrounding the symbol. Next we split the data into training, validation, and test sets randomly.

Once training, validation, and test files are saved with their corresponding bounding boxes and or labels, we were able to train the YOLOv5 model on the split data. Since we had no need to evaluate the accuracy of the bounding boxes for assignment purposes, we only created bounding boxes for the training and validation sets, not the test set.

To train the model we utilized a SLURM script and YAML file required for training with YOLO. The SLURM script ”gpu_job_train.sh”, contains the command to run the YOLOv5 train function and arguments including batch size and the corresponding YAML file. The YAML file, ”eel5840.yaml”, specifies file structure of the train and test data and the corresponding labels for all classes of mathematical symbols. The model is trained using YOLOv5. YOLOv5 uses neural networks for object detection in real time. It uses three primary tools, residual blocks, bounding box regression, and intersection over union (IOU). First it divides the image into a grid that has equal dimensions 32Sx32S. The bounding box, as we mentioned previously, we input along with the images and each box includes a width, height, class label and bounding box center. YOLO first divides the image in N grids. The grids predict the bounding box coordinates as well as the object's label and the probability of the object being present in that box. YOLO uses IOU to predict how much the inputted bounding box (which we assume to be the real bounding box) overlaps the predicted bounding box. This is to ensure YOLO is creating accurate bounding boxes of the image. These bounding boxes are essential in multi-symbol classification, it will help us to determine where in the image each symbol is located so that we can label them correctly. YOLOv5 is one of the newest versions of YOLO and has been pre-trained on the COCO dataset.

Lastly we test our model using the test set of data set aside during the preprocessing stage. In this stage the images are input to our model but not their labels or their bounding boxes. We can use our labels to determine the accuracy of our model.

III. Experiments

Our final experimental design for our project split the data into 64-16-20 train-validation-test split. In addition, we created another dataset of images with multiple symbols for assessing the accuracy of the model; this data was only used for testing. We trained our model with a batch size of 32 and for 1000 epochs using the train.py function within the YOLOv5 repository. In practice, however, our model stopped at around 170 epochs due to lack of performance improvement, and the model output we used was a checkpoint at around epoch 70. Due to the need for YOLOv5 to take in images at increments of 32, we upscaled the images from 300x300 to 320x320 using the Pillow library. The dataset for training the YOLOv5 model is converted to its correct format the train.ipynb code; the images in the train and validation splits are saved to a folder structure for training with the image's bounding boxes and labels saved in a corresponding text file. The image name refers to the image ground truth label followed by its index location in the NumPy array for data training. Bounding boxes were not generated for the test set as assessment of bounding box accuracy in test data was not a goal for this project, instead the images and corresponding labels were saved to a NumPy array for loading and evaluation in test.ipynb.

For ease of generating bounding boxes for our model, we initially assumed that our bounding boxes took up the entire image, however we found low accuracy in testing multiple-symbol images. This is likely due to an amount of images in the class-compiled dataset in which the symbol bounding box is not equivalent to the size of the image. We attempted to rectify this with some additional code implemented in the train.py script to generate bounding boxes through image preprocessing; this was performed from suggestions by Dr. Silva. For each image in the training and validation sets, bounding boxes were generated by applying global Otsu's thresholding to binarize the image, then dilating and eroding the binarized result. This result was inverted. We iterated row-wise to detect the uppermost row and bottom-most row with positive pixels; these represented the top and bottom of the bounding box. This iteration was repeated column-wise to detect the left and right of the bounding box. Once the edges were found, we computed the center of the bounding box in the x-axis and y-axis, as well as the bounding box dimensions, and normalized it with respect to image size. The class, center, and dimensions of the bounding box were saved to the corresponding text file for each image.

Training with these labels resulted in inadequate results on our multi-symbol image dataset, and we believed this to be due to the model not being trained on where in the image space each symbol was. Thus, we augmented data by concatenating randomly selected images from train or validation sets together horizontally, generating bounding boxes using the above procedure, and saving the means with respect to the concatenated image's placement in the resulting image. We generated 300 additional augmented images in training and validation datasets using this procedure.

In terms of experimental results, our model performed with approximately 96 percent accuracy on the test set generated from training data. For our compiled test set of multi-symbol images, our model performed with 62 percent accuracy. With respect to the easy test set, it should be noted that only the first output was used in predictions, meaning that the model may have output more than one label, however only the first was used for evaluation. When the model did not output anything, the test script labeled it as class 10. For the hard test set, we should note the only positive predictions were those included in a result where the prediction label size equals the label size. This means that any prediction results where our prediction size does not equal the label size negatively affects the accuracy.

We also considered what other augmentations could be applied to our dataset other than spatial concatenation. Since some simple scalings and rotations to images could transform an image in one class to another, we decided not to augment the data through spatial transformations. For example, stretching the pi symbol vertically produces the product symbol, while rotating the plus sign 45 degrees produces an x.

IV. Conclusions

In the initial phase of our project, we cleaned our dataset in order for the implementation of the model to yield accurate and precise outputs. There were many mislabeled samples due to a variety of inconsistencies resulting from the symbols being handwritten. One thing we learned from this project is the additional effort required for object detection as opposed to single symbol classification. In this, we saw how object detection requires classification and localization to identify an image from a set of predefined categories whereas single symbol classification only requires image recognition. Overall, we successfully trained our dataset of handwritten math symbols and equations with the use of YOLOv5. We were also able to conclude that object detection requires an accurate bounding box in the training phase to produce accurate labels, and we were able to implement data augmentation and image processing techniques to generate bounding boxes automatically.

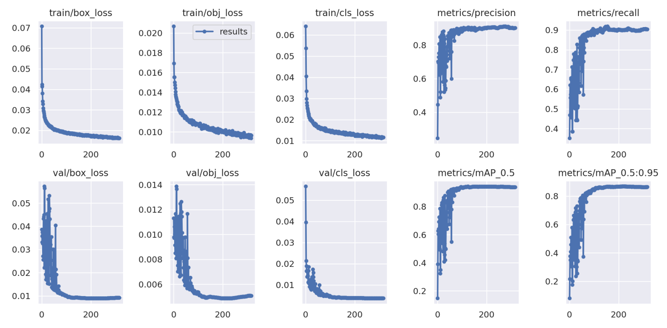

Fig. 3. Output results from model training.

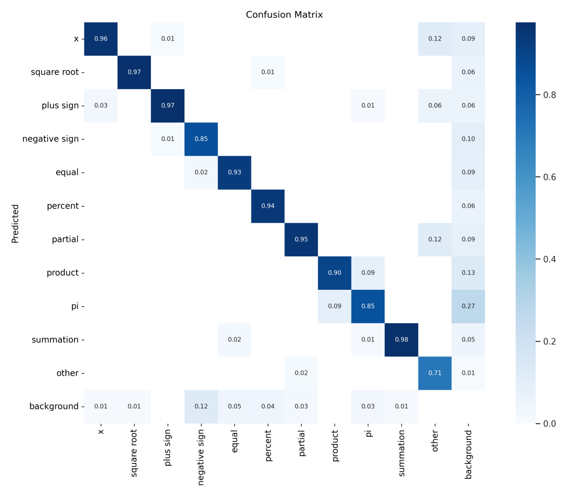

Fig. 4. Confusion matrix of results.

References

[1] G. Yang, W. Feng, J. Jin, Q. Lei, X. Li, G. Gui, and W. Wang, "Face Mask Recognition System with YOLOV5 Based on Image Recognition," IEEE 6th International Conference on Computer and Communications (ICCC), December 2020, pp. 1398-1404.

[2] M. Karthi, V. Muthulakshmi, R. Priscilla, P. Praveen and K. Vanisri, "Evolution of YOLO-V5 Algorithm for Object Detection: Automated Detection of Library Books and Performace validation of Dataset," 2021 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES), 2021, pp. 1-6, doi: 10.1109/ICSES52305.2021.9633834.

[3] S. Karaoglu, et al. "Words matter: Scene text for image classification and retrieval." IEEE transactions on multimedia 19.5 2016, pp. 1063-1076.

[4] Y. Zhu, C. Yao, and X. Bai. "Scene text detection and recognition: Recent advances and future trends." Frontiers of Computer Science 10.1, 2016, pp. 19-36.