Object Detection Computer Vision Project Report

Nehar Poddar, Carter Ithier, Alexander Gear
poddar.ne@northeastern.edu, ithier.c@northeastern.edu, gear.a@northeastern.edu

December 2020

1 Objective and Significance

Object detection is an increasingly popular area of research. It is one of the most challenging aspects of computer vision and recently, the use of deep learning in this field has led to great advances [22]. In object detection, the goal is to create a system that can both classify objects from specified categories in an image, as well as identify a location for such instances of the objects found. Typically, the predicted location is depicted as a bounding box that encloses the object. Object detection algorithms are used in many applications. For instance, they may be used for security and video surveillance, autonomous driving, robotic vision, human-computer interaction, medical imaging, and much more [22, 24]. For example, Chen used object detection for car license plate detection [4], while Lan et al used object detection to detect pedestrians [19], and Al-Masni et al used it for detection and classification of masses in women's breasts [24].

One main problem with current deep learning object detection algorithms is that the majority of them are slow during inference and as such cannot run in real-time while retaining high accuracy [3]. However, one notable algorithm that is both accurate and capable of real-time inference is YOLO: You Only Look Once [30, 28, 29, 3]. In our project, we set out to recreate the YOLO v3 algorithm from scratch using the deep learning framework PyTorch. After implementing it, we tested our object detector on the MS COCO data set, [21], and evaluated its success.

We had several motivations for this project. One was that the topic seems important because of the wide variety of applications it can be used for (as can be seen by the examples mentioned previously). And specifically in regards to applications, all of our team members are interested in robotics or autonomous vehicles which are both fields where object detection is extremely important. Also, considering the field of object detection, YOLO is very exciting because of how fast it can perform inference. It would be great to see this in practice.

During the project, we were able to successfully recreate many data augmentation techniques such as photometric and geometric distortions, cutmix, and mosaic– all of which were used in YOLO v4 [3]. We were also able to implement the YOLO v3 architecture, load pre-trained weights, and make detections on the COCO test set. Lastly, we attempted to write a loss function and training loop and train a model on two different data sets. Although the pre-trained weights performed well on the handful of images we visualized results on, they performed relatively poorly when looking at the statistics of Mean Average Precision (AP) and Recall (AR) for the test set. We performed worse than YOLO v3 and the 10 other models we had data for in almost all categories of AP and achieved an overall Mean Average Precision of only 20.3%. The models we trained did even poorer and had an overall Mean Average Precision of 0% indicating that the model was unable to learn anything.

2 Background

2.1 Previous Work in Object Detection

Object detection is one of the areas of computer vision that is growing and developing very rapidly (mostly due to deep learning). From time to time new algorithms and models are developed which keep outperforming previous ones. Since the creation of AlexNet in 2012, which was a type of Deep Convolutional Neural Network (DCNN) for object classification that broke many records, deep learning models have become the main source of improvement for object detection [22]. There are two main types of object detection models: one where you try to identify an instance of a specific object (such as a specific person's face) and the other where you try to detect instances of previously unseen objects of specific categories (such as dogs or cars) [22].

Image classification involves predicting and assigning a class label to one object in an image, whereas object localization involves identifying the location of, and drawing a bounding box around, one or more objects in an image. Object detection is more challenging and combines these two tasks by drawing a bounding box around each object of interest in the image and then assigning them a class label. Together, all of these problems are referred to as object recognition. Today, there exists a wide range of pre-trained models for object detection such as YOLO, RCNN, Fast RCNN, Mask RCNN, and Multibox [32]. Therefore, it is becoming increasingly fast and easy to detect most objects in a video or image.

There are several standards for evaluating the performance of an image classification model. One is the mean classification error across the predicted class labels. Another compares the distance between the true and predicted bounding box for the expected class. Yet another frequently used evaluation method is computing the precision and recall across each of the best matching bounding boxes for the known objects in the image. These metrics are discussed in more detail in Section 3.3 [6, 10].

Object Detection models are split into two main categories: region-based frameworks (also called two-stage frameworks) and unified frameworks (also called one-stage frameworks) [22]. In two-stage frameworks, region proposals are made for an image that are independent of object categories, features are taken from the proposed regions, and then classifiers that are specific to different categories are used to identify if there are any objects of interest in the proposed region [22]. The primary examples of two-stage frameworks are the R-CNN family.

R-CNN Model Family (Region-Based Convolutional Neural Network) includes R-CNN, Fast R-CNN, Faster R-CNN, and Mask R-CNN which are designed for object localization and object recognition [15].

The R-CNN model is comprised of a region proposal, a feature extractor, and a classifier. It generates and extracts category independent region proposals, extracts features from each candidate region, and classifies features as one of the known classes [13].

The Fast R-CNN model trains in multiple stages. It starts by taking the image and passing it through a deep convolutional neural network. It then interprets the CNN as a fully connected layer and splits the model into two outputs, class prediction and linear bounding box. This process is then repeated multiple times for each region of interest in a given image. The model is significantly faster to train and in making predictions than R-CNN [12].

Faster R-CNN has an architecture called a Region Proposal Network, or RPN, that is designed to both propose and refine region proposals as part of the training process. The regions are then used in concert with a Fast R-CNN model in a single model design. These improvements both reduce the number of region proposals and accelerate the test-time operation of the model to near real-time with what was (at the time) state-of-the-art performance [31].

The most recent model in the R-CNN family is Mask R-CNN which extends Faster R-CNN, allowing for predictions that not only detect objects but also make instance segmentation predictions [14].

In one-stage frameworks, there is no region proposal step. Instead, there is a single pipeline that directly calculates bounding box offsets and class probabilities [22]. Notable examples of one-stage frameworks include OverFeat which pioneered one-stage object detectors by combining feature extraction, location regression, and region classification in the same CNN [33]. YOLO, described in the next section, is also an example of a unified framework. Other examples include CornerNet, which at the time of its publication outperformed previous unified frameworks (although the inference time was extremely slow in comparison), and Single Shot Detector (SDD) which borrows ideas from Region Proposal Networks and multiscale convolutional features but does everything within one network [22].

2.2 YOLO

2.2.1 YOLO v1

Unlike other object detection models that rely on classifiers and region proposals, YOLO is relatively unique in the sense that it is a single convolutional network that treats object detection as a regression problem [30]. It does this by dividing the given input image into an $S \times S$ grid and allowing each cell to predict a fixed number of bounding boxes for an object. The grid cell where the center of the object lies is responsible for the detection of that single object. In addition to bounding boxes, the associated class probabilities are also predicted [30].

Every bounding box prediction consists of five elements: x, y, w, h and a confidence score where (x, y) are the coordinates of the center of the object in the input image (offset relative to the grid it lies in) and w and h are the width and height of the object, respectively. The confidence score is the probability that the box contains the object and tells us how accurate the model thinks the predicted bounding box is. Also predicted is the conditional class probability which is the probability that the detected object belongs to a particular class (one probability per category for each cell) given that the grid cell contains an object [30]. The above description explains the basics of YOLO v1, however, YOLO has greatly evolved since its inception. Its further iterations are described in the following sections.

2.2.2 YOLO v2

YOLO v1 had many shortcomings, such as low localization accuracy and difficulty in recognizing small objects. YOLO v2 became faster and more accurate. It added batch normalization, chose a higher resolution classifier, and also applied Faster R-CNN's idea of anchor boxes to its bounding box regression. It determines the anchor box sizes by using k-means clustering on the training set bounding boxes to better align with object shapes. To solve the problem of model instability after the addition of anchor boxes, YOLO v2 constrains the object center regression predictions by using a logistic activation. Another improvement was the addition of DarkNet-19 for feature extraction. DarkNet-19 was inspired by "Network in Network" [20], GooLeNet's bottleneck structure, and the introduction of multi-scale training which results in different input resolutions at different periods of training. Lastly, to improve the detection of small objects, YOLO v2 added a pass-through layer to merge features from an early layer. One way that it departed from its predecessor is that YOLO v2 experimented with a version that is trained on a 9000 class hierarchical data set, which also represents an early attempt at multi-label classification in an object detector [28].

	Backbone	Detector
Bag of Freebies (BoF)	• CutMix • Mosaic data augmentation • DropBlock • Class label smoothing	• CIoU-loss • Cross mini-Batch Normalization • DropBlock • Mosaic data augmentation • Self-Adversarial Training • Multiple anchors for a single ground truth • Cosine annealing scheduler • Optimal hyperparameters • Random training shapes
Bag of Specials (BoS)	• Mish activation • Cross-stage partial connections (CSP) • Multi-input weighted residual connections (MiWRC)	• Mish activation • SPP-block • SAM-block • PAN path-aggregation block • DIoU-NMS

Figure 1: Breakdown by backbone and detector of examples of Bag of Freebies and Bag of Specials from YOLO v4 [3].

2.2.3 YOLO v3

YOLO v3 only made a few improvements from YOLO v2. Like YOLO v2, v3 uses anchor boxes whose priors are determined using k-means clustering. It continues to use logistic activation to constrain the bounding box object center and predicts the center, height, width, and objectness score for each bounding box. It uses multi-label classification as before. YOLO v3 predicts bounding boxes at three different scales and one of the biggest improvements it made was the use of a new feature extractor called Darknet-53. YOLO v3 got rid of v2's pass-through layers and fully embraced FPN's multi-scale predictions design. YOLO v3 performed similarly to other SSD type detectors, but was 3x faster and, compared to v2, did very well on small object detections (although slightly worse on medium and larger objects). Although decent at predicting bounding boxes, it still had trouble perfectly aligning them with the object [29].

2.2.4 YOLO v4

In experiments, YOLO v4 obtained an AP value of 43.5% on the MS COCO dataset and obtained a speed of 65 FPS on the Tesla V100, beating the fastest and most accurate detectors. When compared to YOLO v3, the average precision increased by 10% and frames per second by 12% [3]. The authors used several new features to make their design suitable for efficient training and detection such as weighted residual connections, cross-stage partial connections (a new backbone that can enhance the learning capability of a CNN), cross mini-batch normalization, self adversarial training (a new data augmentation technique that operates in two forward backward stages), mosaic data augmentation, DropBlock regularization (a better regularization method for CNN), and CIoU loss. The authors of the YOLO v4 paper distinguish the methods that are used to improve the object detector's accuracy and achieve a fast operating-speed neural network into two categories. One category is Bag of freebies (BoF): methods that can make the object detector achieve better accuracy without increasing the inference cost. BoF methods only change the training strategy and not the architecture. An example of BoF is data augmentation, which increases the generalization ability of the model. Minor changes can be made to make photometric distortions such as changing the brightness, saturation or contrast, and adding noise. Geometric distortions can also be made such aas rotation or cropping. The second category is Bag of specials (BoS). BoS are plugin modules and post-processing methods

that only increase the inference cost by a small amount but can significantly improve the accuracy of object detection [3]. A more detailed list of BoF and BoS can be seen in Figure 1.

2.3 Our Project Contribution

As can be seen in the following Methodology section, our methodology was nearly identical to YOLO v3. Our goal for the project was simply to recreate the YOLO v3 paper and see if we could achieve similar results. However, we still feel that this was an interesting project because of the various applications of object detection and the inference speed we obtained despite the fact that our code was written purely in Python. This report also demonstrates how difficult it can be to recreate papers and how papers could be improved to include more information that would aid in their replicating their results.

3 Methodology

3.1 Data set

Two different data sets were used: COCO and a small data set with nuts that was obtained from Tony607's Github repo [11]. The following two sections describe each data set.

3.1.1 COCO

For this project, we trained and evaluated on the Microsoft Common Objects COntext (MS COCO) data set [21]. The entire COCO data set has 2.5 million object instances labeled across 328k images and contains 91 different object categories. COCO is readily available for download and consists of both images and their corresponding annotations [5]. The annotations contain information like the image dimensions, object categories and corresponding IDs, image and annotation IDs, and points consisting of the object segmentation and bounding box. YOLO does not make instance segmentation predictions so we did not use the segmentation information in the annotations; we only used the bounding box information.

Because we had limited time to train and difficulty storing and transferring that many images (which did not include the number of images we wanted to augment) we decided to use a subset of the data. We decided to use only 10 categories from the 2017 train and 2017 test COCO data sets: person, dog, bottle, cup, fork, bowl, dining table, TV, laptop, and book. We picked categories that had large numbers of pictures, varying difficulty levels in detection, and were common-place objects so we could test our model on pictures taken in our homes.

There were four different subsets of the COCO data set that we used. The number of overall images, annotations, and categories can be seen in Table 1, the number of images by category can be seen in Table 2, and the number of annotations by category can be seen in Table 3. In the tables, the second column is "Reduced COCO." This subset of the COCO data set was obtained by separating out all pictures and annotations that had our categories of interest. There was an overwhelming number of annotations with people that led to class imbalance so we removed all images from this subset that only contained people. The "Subset of Reduced COCO" consists of a random selection of 1000 photos taken from the previously mentioned data set. This was done because we were having difficulties in training the network and we were rushed for time so in order to be able to iterate quickly we decided to have a smaller set to train on. The "Augmented Data Set" consists of the "Reduced COCO" subset along with an additional 28,068 augmented images that were produced following the methodology described in Section 3.2.1. Lastly, the "Test Set"

	Reduced COCO	**Subset of Reduced COCO**	**Augmented Data Set**	**Test Set**
# Images	32,579	1,000	60, 647	3,450
# Annotations	172,733	5,235	332,082	160,84
# Categories	10	10	10	10

Table 1: A break down of the number of images, annotations, and categories in the various train and test sets we created that utilized COCO.

Category	**Reduced COCO**	**Subset of Reduced COCO**	**Augmented Data Set**	**Test Set**
person	15,243	451	31,286	2,693
dog	4,385	133	13,365	177
bottle	8,501	272	15,474	379
cup	9,189	299	19,165	390
fork	3,555	101	11,249	155
bowl	7,111	236	15,655	314
dining table	11,837	353	25,326	501
TV	4,561	141	13,349	207
laptop	3,524	112	13,832	183
book	5,332	164	12,228	230

Table 2: The number of images that each category had in the various training and test sets we created from COCO.

consists of the 2017 COCO Validation Set (called Val2017 on the COCO website [5]), but which has been filtered to only have images and annotations from our classes of interest.

We chose COCO for several reasons. Firstly, it is one of the standard benchmark data sets used to evaluate object detection models. This allowed us to easily compare our results to those reported by other papers (including YOLO v2, v3, and v4). Also, because it consists of many common objects such as dogs, eating utensils, laptops, etc, we would be able to easily perform inference on new photos that we take with such objects in them. Lastly, there exists a Python COCO API [7], which allowed us to easily visualize prediction results and evaluate the performance of our model.

3.1.2 Nuts Data Set

The fruits nuts segmentation data set is a data set used in several online tutorials to train object detection models such as Facebook Detectron 2 [35]. It can be found on Github here [11]. It consists of 18 images and 3 classes: hazelnut, fig and date. A breakdown of the number of images and annotations overall can be seen in Table 4 and the breakdown by category can be seen in Table 5. We used it to test our model because it is a small data set, which made it was much faster to train than our COCO data set. It was helpful to check and debug our model before training it on the Discovery cluster for our large data set.

Category	Reduced COCO	Subset of Reduced COCO	Augmented Data Set	Test Set
person	52,257	1,533	88,245	10,777
dog	5,500	165	17,014	218
bottle	24,070	737	36,283	1,013
cup	20,574	642	37,428	895
fork	5,474	150	15,110	115
bowl	14,323	435	28,272	623
dining table	15,695	443	33,510	695
TV	5,803	167	17,660	288
laptop	4,960	183	18,853	231
book	24,077	780	39,707	1,129

Table 3: The number of annotations for each category in the various training and test sets we created using COCO.

Number of Annotations	164
Number of Images	18
Number of Categories	3

Table 4: Number of annotations, images, and categories in the fruit nuts data set.

	Hazelnut	Fig	Date
Number of Images	17	16	17
Annotations	53	40	71

Table 5: Breakdown by category of the number of images and annotations for the fruit nuts data set.

3.2 Methodology

3.2.1 Data Augmentation and Pre-processing

Before being fed into the neural network, all images from our data sets were resized to be 416×416, and the pixel RGB values were normalized to lie in between 0 and 1.

A total of 28,068 images were augmented from the "Reduced COCO" data set resulting in the size of our data set nearly doubling. The types of augmentations (which are described in the following sections) were standard types such as photometric and geometric distortions as well as two less commonly used ones described in the YOLO v4 paper– cutmix and mosaic [3]. All code was done in Python and utilized the computer vision library OpenCV. We attempted to create and maintain class balance across all 10 categories in the final augmented data set. We did this by picking pictures to augment based on a probability weight distribution we determined. The weights in this distribution summed to one and pictures with fewer annotations had more weight (and thus were more likely to get picked) than pictures with more annotations. Included in the weight was a measure of how frequent a category was in an image. For instance, in many images with people, there were multiple people in one image whereas for dining tables there was typically only one dining table per image. Thus, categories with a smaller frequency per image had higher weights. This still was not a perfect solution because each image could have multiple annotations of different categories per image which meant that augmenting images for a class like TV could also result in more annotations of people if there are people in the image with the TV. However, as can be seen in Table 3 we were able to get the number of annotations within six times of each other.

Industry Standard Augmentations Based on the description of common data augmentation practices in [29] and [34], we created an augmentation pipeline implementing such features. We initially tried to use a Python library called Albumentations to augment our data because it had all of the features that we wanted in our pipeline [1]. However, we found that when generating over a few hundred photos, the library would start having extremely inaccurate bounding boxes for the augmented image. Examples of such cases can be seen in the two right-most images in Figure 2. Another problem we found with bounding boxes was that the library created the new bounding box based on the original. However, results were much more accurate if you transformed the mask of the image (because the COCO data set has segmentation information) and then recalculated the new bounding box based on that transformed mask. As a result, we were unable to use any augmentations from the library that could result in different bounding boxes in the output image.

To remedy this, we created our own pipeline and made the new bounding boxes it created based on the mask of the transformed image. In our pipeline, we used some features from Albumentations that did not affect bounding boxes such as FancyPCA (based on Krizhevsky's paper [18]), color jitter (which randomly changes the brightness, contrast, and saturation of an image), blur, the addition of gaussian noise, and a method that changes the hue, saturation, and value of the input image. The features that we implemented ourselves were resize, horizontal flip, a random crop and re-scaling (which is required to contain at least one bounding box in the new image), rotation, and Cut Out. Cut Out was introduced by DeVries et al and is when you randomly replace small square sections in an image with all black [8]. In our pipeline, one of Fancy PCA, color jitter, or hue saturation value change were applied with 50% probability. Blur and gaussian noise were applied with 30% probability. Horizontal flip was applied with 50% probability, Cut Out was limited to a maximum of 3 squares to be converted to black and was applied with 33% probability, random crop and re-scaling were applied with 20% probability, and a rotation between $-15°$ and $+15°$ was applied with 15%

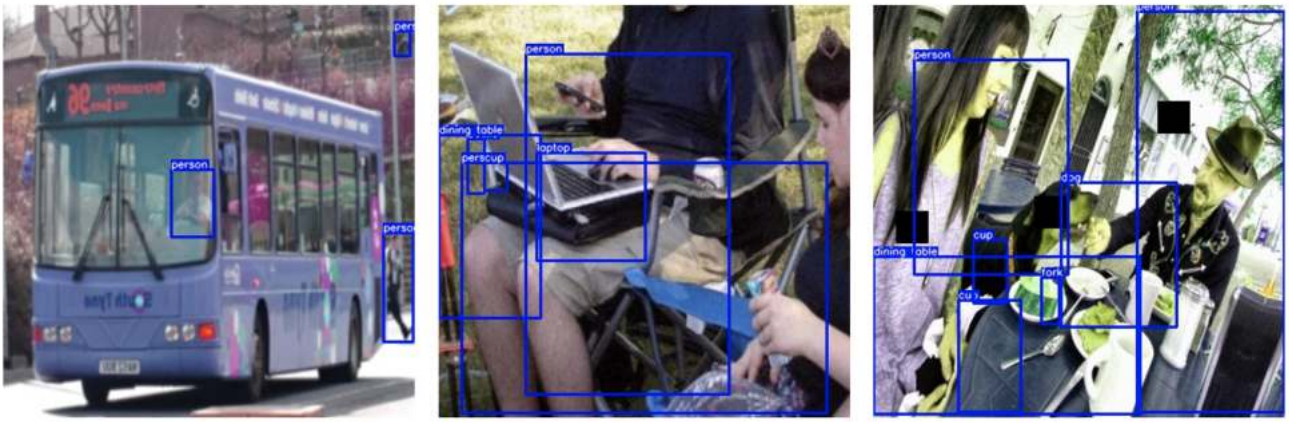

Figure 2: Ground truth bounding boxes produced by the Albumentations data augmentation library were sometimes correct such as in the left-most image, but others were very inaccurate such as the following two images.

probability. Pictures were only chosen to be augmented if the segmentation masks for each object instance were at least 300 square pixels in area.

Cutmix and Mosaic Augmentations The other two types of data augmentation we performed were cutmix and mosaic [36, 3]. In both of these methods, multiple images are cropped and combined into one new image. For cutmix, two images are combined and for mosaic it is four. Both of these methods were used in the YOLO v4 paper in which the authors argue that such methods allow for the network to learn representations of objects "outside of their normal context" which aids in training [3]. An example of outputs for each algorithm can be seen in Figures 12 and 13.

The algorithm we developed that implements cutmix can be seen in Algorithm 1. Lines 1-5 initialize several variables for the algorithm. The variable *size* is the dimension of the input and output image. In our case, we had scaled all of our data to be 416×416. The variables min_area and min_intersect_percentage are parameters that may need to be fine-tuned further in the future. They specify what area an instance must have to be counted in the annotations and the minimum allowed percentage that the bounding box area for the cutmix image should have compared to the original bounding box in the inputted image. We chose these values after much experimentation in which we visualized bounding boxes on the images output by our cutmix and mosaic algorithms and determined whether or not the bounding box seemed reasonable for a network to learn and should be included or excluded. Additionally, the variables min_percentage and max_percentage, which control the minimum and maximum width of the original image in the new cutmix image, were chosen somewhat arbitrarily until we got results that we thought looked similar to the augmentations shown in [3]. As such, there is room for changing these values.

Next, we initialize the cutmix image we are going to return in line 6. In lines 7-10 we check that all areas for each instance in the original objects are greater than the minimum area specified in line 4. If they are, we continue with the algorithm, otherwise, we return None to indicate failure. In lines 11 to 14 we calculate how many rows and columns the left side of the cutmix image should have as well as the right side. Then in lines 15-20 we obtain ROIs (Regions of Interest) for the left and right-hand side of the cutmix image and copy the pixels of those ROIs to the appropriate portions of the cutmix image to be returned. In order

to obtain the ROIs, we randomly sample points near at least one bounding box that exists in the original image. This is done inside the "determine_roi" function. In the function, we treat that point as one corner of a rectangle of the appropriate dimensions. If this rectangle ends up containing at least one bounding box, then the rectangle rows and columns are returned. Otherwise, a new point is sampled and the process is repeated until success.

In lines 21-37, new annotations are created for the cutmix image. This is done by creating a mask for each object instance and cropping and pasting the appropriate ROIs of the mask based on what happened for the real images. We are able to create the masks because the original COCO data set contains segmentation information for each object instance. Next, new bounding boxes are determined from the mask and if they are above the minimum area and consist of a large enough percentage of the original bounding box, then the annotation is saved. Once this is done, if the length of new annotations is greater than 0 then the new cutmix image and annotations are returned, otherwise, None is returned to indicate failure.

The algorithm we developed for mosaic is almost identical to Algorithm 1, except $y1$ and $y2$ are different values that sum to $size$ and the overall process is done for four images and their respective Regions of Interest (ROI) instead of two. In the mosaic algorithm, $y1$ and $y2$ are calculated in the same manner as $x1$ and $x2$ (they are within the range min_percentage $\times$ size and max_percentage $\times$ size).

Post-Processing In earlier iterations of creating the data set, we had an extremely unbalanced data set due to the number of annotations that involved the person class. This resulted in some annotation numbers being 52 times that of less frequent classes. To combat this, we wrote a post-processing script that would slowly eliminate images in the data set one at a time until certain criteria were met in order to create a more balanced data set. Pseudo-code for this algorithm can be seen in Algorithm 2. In the algorithm, we considered the data set to be balanced if the number of annotations for each class is within ten times each other, and otherwise, it is considered not balanced. We run the algorithm until either the data set is balanced or a maximum number of iterations has elapsed (which we specify as four times the number of images in the data set). We repeatedly get a list of category ids that are not balanced (because their number of annotations are too large) and then get all of the image ids that contain at least one of the category ids found in the previous step. Then annotations are obtained for one of the image ids that are randomly sampled and it is determined which category id has the maximum number of annotations in that image. If the maximum category id found is one that belongs to the list of category ids that have too many annotations, then this image is removed from the data set. We found that it was important to only remove the image based on the maximum category id. That way, we were more likely to avoid removing images that may have other annotations in it whose numbers were too small.

3.2.2 Architecture

YOLO v3 works by splitting each 416×416 input image in a batch into an $S \times S$ grid where S can take on three different values ($S_1 = 13$, $S_2 = 26$, and $S_3 = 52$). $B = 3$ anchor boxes (priors) are chosen for each grid cell using k-means clustering on the dimensions of the training set's bounding boxes.

For YOLO v3, $k = 9$ (chosen arbitrarily) [29], and the clusters are then divided up evenly among three different scales (described in the following paragraph) leading to three bounding box predictions for each cell at each of three different scales.

Predictions are made for each anchor box as sets of four coordinates to indicate the x and y position of the center of the predicted object (relative to the top left corner of the object's respective grid cell and passed

Algorithm 1: Our Cutmix Algorithm

Input: Two images and corresponding annotations

Result: Two Images Merged into One Image with Corresponding Annotations

```
size = 416;
min_percentage = 0.25;
max_percentage = 0.75;
min_area = 200;
min_intersect_percentage = 0.2
cutmix_img ← initialize size × size black image;
img1_areas ← areas for each object instance in image 1;
img2_areas ← areas for each object instance in image 2;
if min(img1_areas) < min_area or min(img2_areas) < min_area then
    return None;
min_cols = min_percentage * size ;
max_cols = max_percentage * size + 1;
x1 ← random integer between min_cols and max_cols;
x2 = size - x1;
y1 = y2 = size;
left_roi_rows, left_roicols = determine_roi(x1, y1, size, img1_anns);
left_roi_img = get_roi(img1, left_roi_rows, left_roi_cols);
cutmix_img[0:y1, 0:x1] = left_roi_img;
right_roi_rows, right_roi_cols = determine_roi(x2, y2, size, img2_anns);
right_roi_img = get_roi(img2, right_roi_rows, right_roi_cols);
cutmix_img[0:y2, x1:] = right_roi_img;
for each annotation for img 1 do
    cutmix_mask ← initialize size × size black image;
    mask ← create mask for annotation ;
    left_roi_mask = get_roi(mask, left_roi_row, left_roi_cols);
    cutmix_mask[0:y1, 0:x1] = left_roi_mask;
    bbox ← get bounding box from cutmix_mask ;
    intersect_percentage = bbox_area / original_bbox_area;
    if bbox_area > min_area and intersect_percentage > min_intersect_percentage then
        save new bbox to annotations;
end
for each annotation for img 2 do
    cutmix_mask ← initialize size × size black image;
    mask ← create mask for annotation ;
    right_roi_mask = get_roi(mask, right_roi_row, right_roi_cols);
    cutmix_mask[0:y2, x1:] = right_roi_mask;
    bbox ← get bounding box from cutmix_mask ;
    intersect_percentage = bbox_area / original_bbox_area;
    if bbox_area > min_area and intersect_percentage > min_intersect_percentage then
        save new bbox to annotations;
end
if number of new annotations > 0 then
    return cutmix_img and new annotations
else
    return None
```

Algorithm 2: Creating a balanced data set

```
Result: A balanced data set
steps = 0;
max_steps = 4 * number_of_images;
while not_balanced and steps < max_steps do
    cat_ids ← category ids that are not balanced;
    img_ids ← image ids that contain at least one of the cat_ids;
    id ← a random id from img_ids;
    anns ← annotations for id;
    max_id ← category id that appears the most in anns;
    if max_id in cat_ids then
        Remove image from data set;
        Update annotation by category numbers;
    step++;
end
```

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_n}
\end{aligned}
$$

Figure 3: Predictions t_x, t_y, t_w, t_h are converted into final bounding box coordinates b_x, b_y, b_w, b_h via these equations where p_w and p_h are the width and height of the priors and c_x and c_y are the grid offsets [29].

through a sigmoid function to obtain a value between 0 and 1) and the height and width of the bounding box surrounding the object.

In addition to the coordinates, an 'objectness score' or confidence is predicted specifying the probability that the anchor box in question in fact contains an object and is calculated as Pr(object) $\times$ IoU(bounding box, object) [28].

Furthermore, C independent logistic classifiers, one for each category (or class) in the data set, calculate the probability that the predicted object belongs to that category. In the original YOLO v3 paper $C = 80$. We initially attempted to make this a parameter that we could set at run time to try $C = 10$. Unfortunately, this caused errors in the program due to mismatching tensor sizes, which would require us to experiment with changing other parameters of the convolutional layers and in the interest of time, we decided to stick with $C = 80$.

The objective function is optimized by the mean squared error between the predicted vector and the ground truth vector (see Figure 7 for details). To improve the detection of objects of different sizes, the network divides the image up into cells at three different scales to predict small, medium, and large objects and then concatenates predictions at the different scales together for the final output leading to a final prediction tensor of $[(S_1 \times S_1) + (S_2 \times S_2) + (S_3 \times S_3)] \times [B \times (4 + 1 + C)]$ [29].

In total YOLO v3 uses 106 layers in its network, including convolutional layers, upsampling layers, skip connections, and detection layers (see Figure 5 and Figure 6 for details). We constructed the network using PyTorch's nn.ModuleList() class and appended layers to it one by one. While all the YOLO guides that we

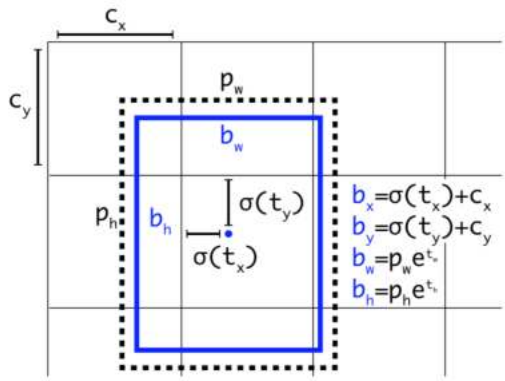

Figure 4: Bounding boxes with dimension priors and location prediction taken from [29].

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 5: Darknet-53 network taken from [29].

used for assistance built their networks from a config file, we decided that both in order to differentiate our code and to understand the network better we would build the network manually [25, 16]. Although this was tedious and led to less clean/modular code, it definitely helped clarify and make obvious the structure of the network.

Using scales of 13×13, 26×26, and 52×52, the output tensor of the final layer for an image is 10647×85. To reduce the total number of detections in the final output we used a utility function to prune the output to a smaller number of "true detections". This is accomplished, first by removing any detections whose objectness score is less than 0.5 and then by using non-maximum suppression, where if two detections are of the same class and have an IOU of more than 0.4, we remove the detection with the lower objectness score.

3.2.3 Loss Functions

We found that the loss function discussed in YOLO v3 was described in too little detail in the paper for us to fully understand and implement it. As a result, we decided to use the loss function from YOLO v1.

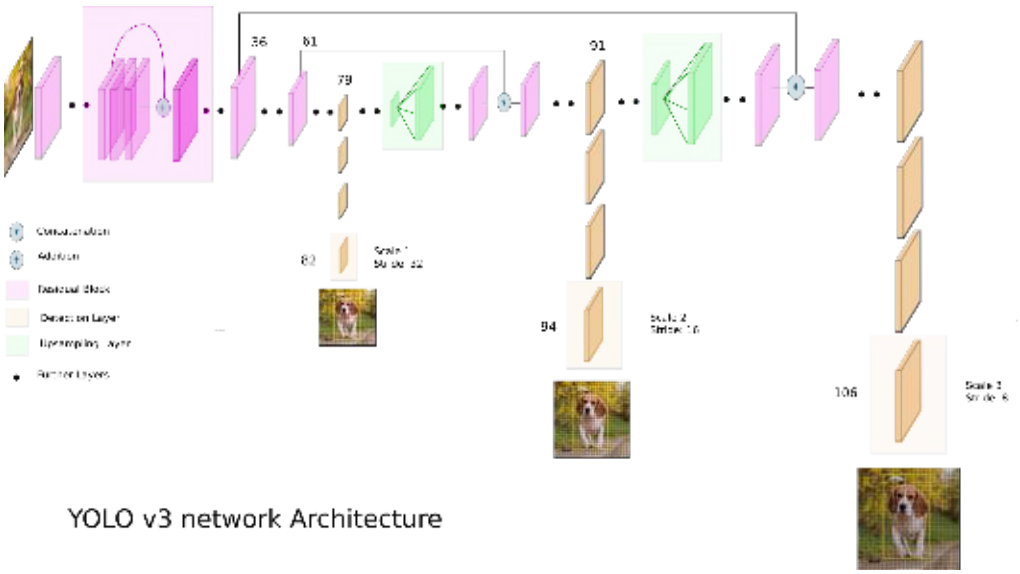

Figure 6: Yolo 3 architecture taken from [17].

$$
\begin{aligned}
\textbf{MSE} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 \\
+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i}^{noobj} (C_i - \hat{C}_i)^2 \\
+ \sum_{i=0}^{S^2} \mathbb{1}_{ij}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

Figure 7: Multi-part Mean Squared Error taken from [30]. λ_{coord} and λ_{noobj} are parameters used to increase the loss from bounding box predictions and decrease the loss from confidence predictions for boxes with no objects. $\mathbb{1}_{i}^{obj}$ means an object appears in cell i and $\mathbb{1}_{ij}^{obj}$ means that cell i's jth bounding box predictor is responsible for the prediction.

A slightly modified version of this loss function described in [30] can be seen in Figure 7. As can be seen, the loss consists of a series of sums of square errors for different terms. In the equation, $\mathbb{1}_{ij}$ is an indicator variable that is 1 when the jth bounding box predictor (of which there are 3 total for each scale trained at) in cell i is "responsible" for the prediction and 0 otherwise. The paper specifies that a bounding box is "responsible" for the prediction if it has the highest IOU of any of the predictors in that particular grid cell. Because it is an indicator function, the loss for most terms only penalizes errors for some bounding box priors, not all. However, for the no object loss, we have $\mathbb{1}_i$ so we are penalizing across all predictors. In the paper, the authors use $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$ and we did the same. The authors explain that they do this to increase the loss from bounding box predictions and decrease the loss from confidence predictions for grid cells with no objects (because most grid cells will not have an object). Other things that should be mentioned is that the bounding box coordinates are normalized to lie between 0 and 1 (x and y are normalized by grid cell location offset and w and h are normalized by image width and height). Also, note that the square root of width and height are taken in the equation to make sure that small differences in large bounding box predictions matter less than in small bounding boxes.

3.2.4 K-means to Pick Anchor Box Priors

YOLO v2 introduced the concept of using anchor box priors in order to predict bounding boxes which resulted in an increase in recall performance [28]. Rather than selecting the priors by hand, the authors used k-means on the bounding boxes in the training set. To maximize good IOU scores and discourage larger boxes from generating more error than smaller boxes, Redmon et al used the following distance metric and found the best trade-off in error and complexity by using 9 anchor boxes:

$$d(box, centroid) = 1 - IOU(box, centroid) \tag{1}$$

To follow in these footsteps, we ran k-means on our augmented data set with $k = 9$. The points consisted of $(width, height)$ and when doing the IOU calculation, we considered the center of all rectangles for those two attributes to be the same. Our stopping criteria was whether no assignment changes were made or whether 1000 steps had elapsed. We found in our experiment that it did not converge in 1000 steps. To increase our chances of finding better clusters, we used the k-means++ algorithm as the method for choosing our initial cluster centroids. A visualization of the results can be seen in Figure 8. The red points are the different bounding boxes from the training set (which was the augmented data) and the black x's are the centroids that were found. These centroids, $(width, height)$, were: (122, 109), (31, 76), (206, 313), (375, 354), (250, 149), (79, 48), (119, 245), (58, 154), (18, 27). We did not end up using these priors in our final model because we did not train on the augmented data set. Instead, we used the ones proposed in the YOLO v3 paper: (10, 13), (16, 30), (33, 23), (30, 61), (62, 45), (59, 119), (116, 90), (156, 198), (373, 326) [29].

3.2.5 Training Set-Up

Our training loop used Adam optimization with a weight decay of 1e-4. We initially used a learning rate scheduler to gradually reduce the learning rate over time, but found that our settings for which epochs the learning rate should decrease did not work well. As a result, we experimented with the learning rate and would change it manually if we saw that the loss was either increasing or not changing. The learning rates we ultimately used ranged from 1e-5 to 5e-7. For training the nuts data set, our initial weights were the pre-trained weights obtained from [27]. We hoped that by starting with this initialization, we would get

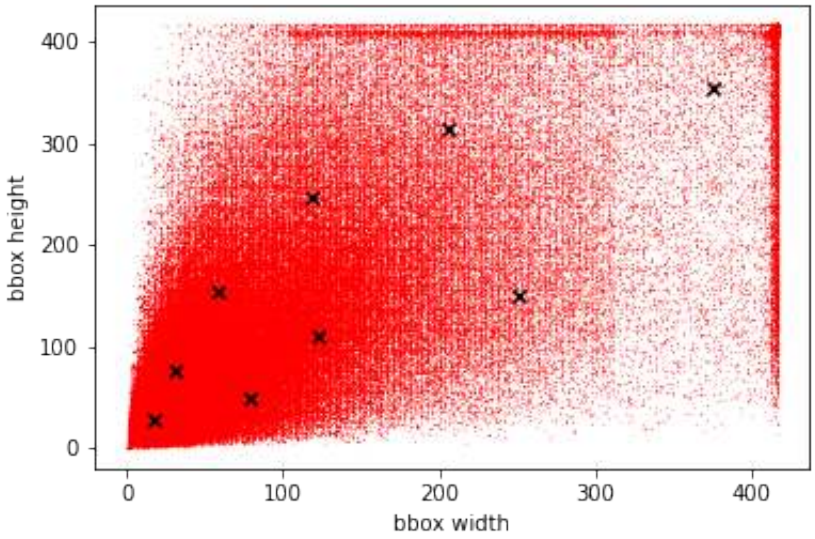

Figure 8: K-means results visualized for finding anchor box priors.

closer to the optimal weights for the nuts after training. For training the COCO data set, we used Xavier initialization for our weights.

We did all of our training on the Discovery Cluster. We tried a few different hardware configurations during training while learning how best to use Discovery. Initially, we used the default GPU node which gave us 128 GB of RAM. This severely limited our batch size. Ultimately, we settled on a k80 GPU node with 500 GB of RAM which allowed for a batch size of 16 and completed roughly 30-50 epochs per hour. Since the maximum consecutive time we were authorized to run a discovery GPU node was eight hours, we could not complete our desired number training epochs in a single session. In order to get around this limitation, we used PyTorch's checkpointing system to save the model parameters and epoch number every five batches. That way, after our time ran out, we could restart close to where we left off.

We trained the "Subset of Reduced COCO" data set for 1000 epochs and the nuts data set for 1500 epochs.

3.3 Evaluation Strategy

One statistic we collected was the frames per second we could achieve while doing inference. In the Results Section we compare this to what the YOLO papers reported.

The true metrics we used to characterize the performance of our model are the standard 12 metrics used to evaluate the performance of an object detection model on COCO [6]. These metrics include different breakdowns of mean average precision and mean average recall. A list of the metrics can be seen in Figure 9 and an explanation of them is below following the descriptions in [10, 9, 2, 26]. Note the AP and AR metrics shown in Figure 9 are averaged over all categories so a more accurate description of these metrics is "mean average precision" (mAP) and "mean average recall" (mAR). Also, it should be noted that the area numbers used to determine the small, medium, and large scales in Figure 9 are measured as the number of pixels in the segmentation mask.

```
Average Precision (AP):
  AP                % AP at IoU=.50:.05:.95 (primary challenge metric)
  AP^IoU=.50        % AP at IoU=.50 (PASCAL VOC metric)
  AP^IoU=.75        % AP at IoU=.75 (strict metric)
AP Across Scales:
  AP^small          % AP for small objects: area < 32^2
  AP^medium         % AP for medium objects: 32^2 < area < 96^2
  AP^large          % AP for large objects: area > 96^2
Average Recall (AR):
  AR^max=1          % AR given 1 detection per image
  AR^max=10         % AR given 10 detections per image
  AR^max=100        % AR given 100 detections per image
AR Across Scales:
  AR^small          % AR for small objects: area < 32^2
  AR^medium         % AR for medium objects: 32^2 < area < 96^2
  AR^large          % AR for large objects: area > 96^2
```

Figure 9: The 12 metrics used in the COCO competition to evaluate the performance of an object detector on the data set, taken from [6].

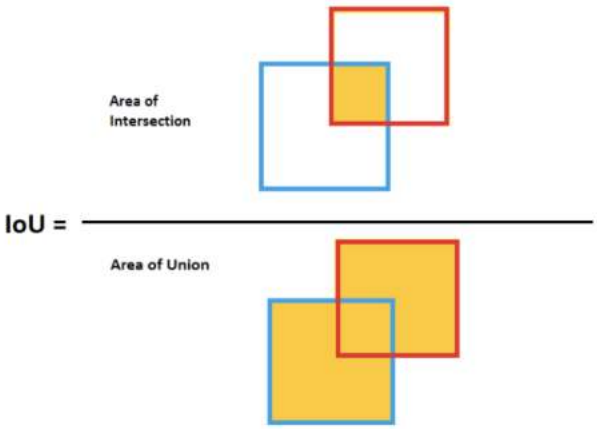

Figure 10: A visual description of Intersection over Union (IoU) taken from [26].

Before going further into the description of these metrics, it is helpful to go over a few terms first. The first is a confidence score. A confidence score is the probability that the anchor box in question contains an object of interest. Second is Intersection over Union (IoU). Given a ground truth bounding box (B_{gt}) and a predicted bounding box (B_p), the IoU measures the overlap between the two by calculating the area of the intersection of the two boxes divided by the union of the two boxes:

$$IoU = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \tag{2}$$

A visualization of IoU can be seen in Figure 10. To determine whether there should be a prediction made, first the IoU is calculated and then a threshold is set so that if the IoU value is above this threshold it is considered a positive prediction and below which there is not considered to be a prediction.

Now we can define a few more terms. If an object classification is correctly made it is a true positive (TP). If there is no prediction for an object and there is also no ground truth object it is a true negative

(TN). If there is a ground truth for an object, but one was not predicted it is a false negative (FN) and if there is a prediction for an object when in fact there was no ground truth for an object it is a false positive (FP). With these definitions we can now describe the metrics precision and recall:

$$precision = \frac{TP}{TP + FP} = \frac{true \quad object \quad detection}{all \quad detected \quad boxes} \tag{3}$$

$$recall = \frac{TP}{TP + FN} = \frac{true \quad object \quad detection}{all \quad ground \quad truth \quad boxes} \tag{4}$$

It can be difficult to create a model that has both high precision and recall and thus there is often a trade-off between the two metrics. One metric that is a single number that captures both precision and recall is called the Average Precision (AP). Average Precision is the area under the precision recall curve as evaluated over a set of 11 spaced recall levels:

$$AP = \frac{1}{11} \sum_{r \in (0, 0.1, \ldots, 1)} p_{interp}(r) \tag{5}$$

where $p_{interp}(r)$ is the maximum precision that had been measured for a corresponding recall that is larger than r:

$$p_{interp}(r) = max_{\hat{r}:\hat{r} \geq r} p(r) \tag{6}$$

To calculate the mean average precision we simply follow the following formula:

$$mAP = \frac{1}{\#classes} \sum_{c \in classes} AP[c] \tag{7}$$

The above equation is fixed for using a specific IoU threshold to determine what counts as a positive prediction. Many of the COCO metrics described above average the mAP metric over IoU threshold's ranging from 0.50 to 0.95 in 0.05 increments giving the following equation:

$$mAP^{IoU=0.5:0.05:0.95} = \frac{mAP_{0.50} + mAP_{0.55} + \ldots + mAP_{0.95}}{10} \tag{8}$$

Equation 8 describes AP^{small} when applied to small objects, AP^{medium} when applied to medium objects, and AP^{large} when applied to large objects, and simply mAP when considering all sized objects. The size category is determined based on the area of pixels in the segmentation annotation of the object instance. Equation 7 applies to $AP^{IoU=0.50}$ when the IoU threshold is set to 0.50 and $AP^{IoU=0.75}$ when the IoU threshold is set to 0.75.

The other metrics in Figure 9 are related to the mean Average Recall (mAR). This is broken up by object size as well as by the number of detections per image. Average recall is the recall over all IoU thresholds in the range [0.5, 1]. It can be calculated as:

$$AR = 2 \int_{0.5}^{1} recall(x) dx \tag{9}$$

where x is the IoU and $recall(x)$ is the corresponding recall. The above equation can be used to calculate the per-class AR. To determine the mean Average Recall that is described in Figure 9 you can use the following equation:

$$mAR = \frac{1}{\#classes} \sum_{c \in classes} AR[c] \tag{10}$$

Statistics of model performance on the majority of the above metrics are presented in [28, 29, 3], thus allowing us to directly compare our results to those of the original papers. Specifically, we compare the

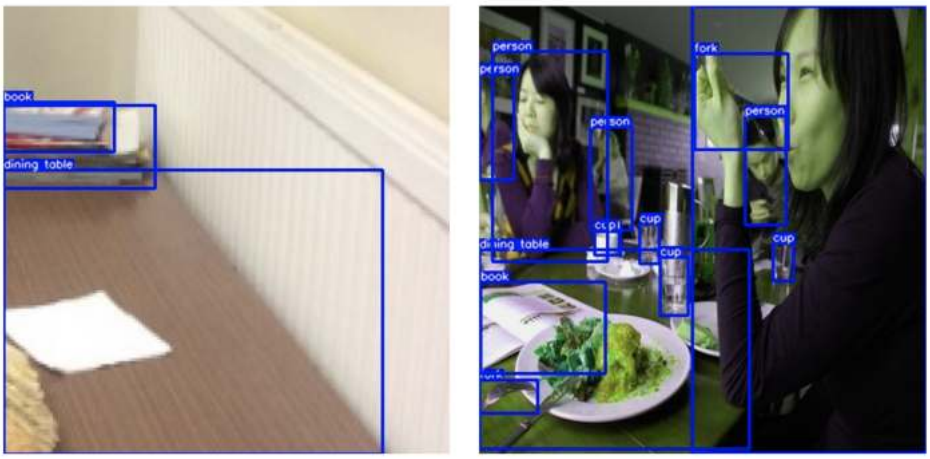

Figure 11: Augmentation results when applying common augmentation techniques (such as geometric and photometric distortions) to our training data.

AP, $AP^{IoU=0.50}$, $AP^{IoU=0.75}$, AP^{small}, AP^{medium}, and AP^{large} results to the YOLO v2 and v3 papers, as well as a handful of other models (including Faster R-CNN with FPN, Faster R-CNN with TDM, DSSD513, and RetinaNet). In following the standard set by the COCO competition, we treat the mAP (which is averaged across all 10 IoU thresholds and all categories) as the most important metric when considering our performance.

4 Results

This section presents our augmentation results, results on the COCO test set using both pre-trained weights and weights we obtained by training ourselves, as well as results on the nuts data set. The majority of the details regarding the algorithms used, set-up of our experiments, and our evaluation metrics are discussed in Section 3. Anything omitted from that section is detailed below.

4.1 Augmentation Results

Figures 11 - 13 showcase some of the augmentation results we achieved. Figure 11 has results from the augmentation pipeline that performed augmentation techniques that were standard in industry such as random cropping, horizontal flipping, etc. In the left image of the figure, random cropping and rescaling have been applied. In the right image, a horizontal flip and color channel changing was applied. Figure 12 shows three examples of outputs of when the cutmix technique was applied to our data and Figure 13 shows examples when mosaic was used.

It is unclear whether such augmentations would be beneficial or harmful to our results as we were unable to successfully train on any data set (as is discussed in later sections). On the one hand, augmentations should help in obtaining a better model because more data is typically good for deep learning models. Also, doing such augmentations is now standard practice in the deep learning field. However, it is possible that the cutmix and mosaic techniques result in objects being too much out of context or view in the new image; this is more of a concern with mosaic than cutmix since it is four images being combined. An example of such a

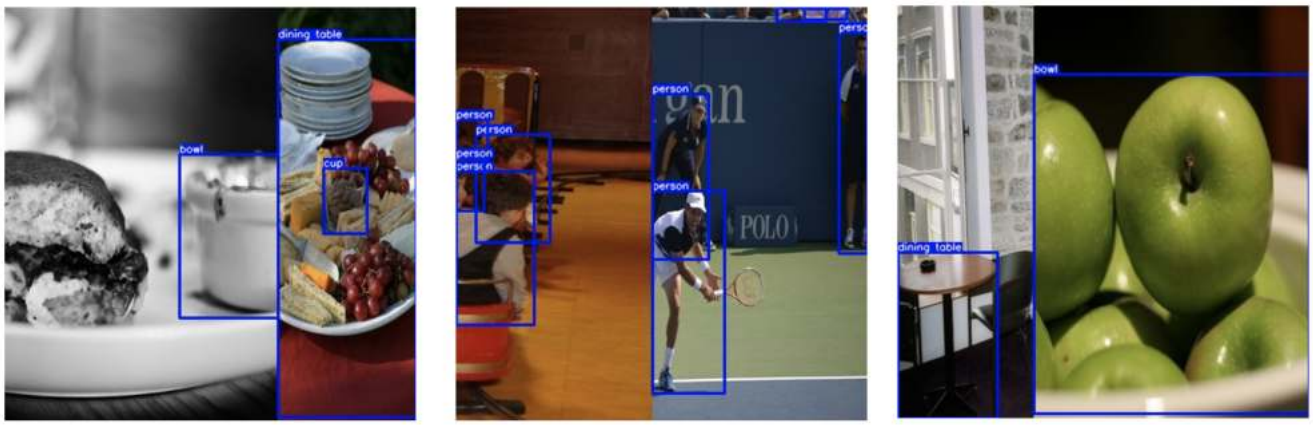

Figure 12: Augmentation results when applying the cutmix technique to our training data.

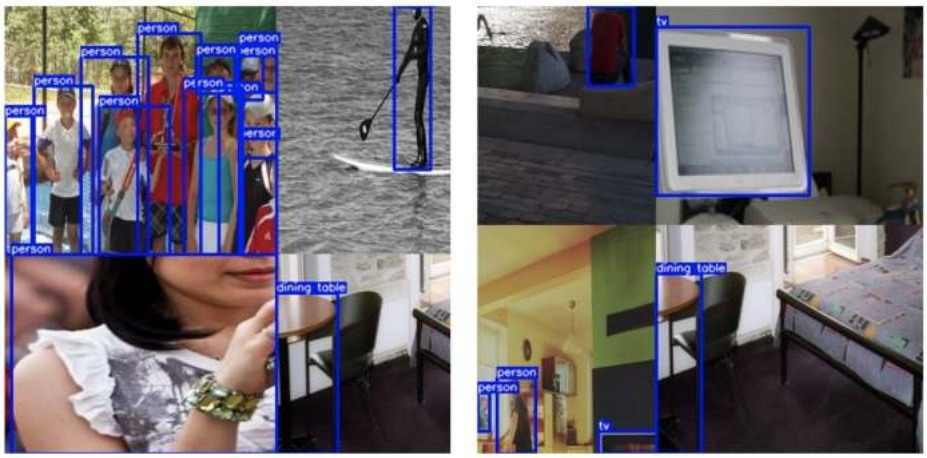

Figure 13: Augmentation results when applying the mosaic technique to our training data.

case is the person in the top left square of the right-most image in Figure 13. Half of the human has been obscured during the cropping process making it difficult to distinguish the person. The same is true for the people in that figure in the bottom left-hand corner. As a result, training a model on such instances may in fact be harmful. The results from a model successfully trained on the augmented data set is needed to draw further conclusions and determine whether the cutmix and mosaic algorithms we created need adjustment.

4.2 Nuts Results

We trained on the fruit nuts data set for a total of 1500 epochs. The training loss was logged every epoch and can be seen in Figure 14. For epochs 1-1000, we used a learning rate of 1e-6 (the orange line in the graph). We noticed that the loss curve was flattening out so we decided to increase the learning rate to 1e-5 for epochs 1000-1500 (the blue line in the graph). Our model was hard-coded to predict 80 classes, even though for this particular data set there were only three classes. As a result, we filtered the prediction output to only correspond with the class indices that matched our truth labels during training (0, 1, and 2).

The Average Precision and Average Recall results for the model we trained on this data set (after the predictions were filtered) can be seen in Figure 15. The -1 values in the table for AP_S and AR_S indicate that there were no ground truth objects that fit these settings, i.e. there were no ground truth annotations for objects with areas smaller than 32^2 square pixels. As can be seen in the figure, every category except those two values that were -1 had a value of zero. This indicates that we did not have a single true positive detection.

Visualization of detections on images in the data set (the same images we trained on because there was no test set available) can be seen in Figure 16. The images pictured in the figure were the four out of the six images that had any detections after filtering by class. The two other images with detections erroneously classified parts of the textured background in the image as one of the fruit nuts. The remaining 11 images had no detections from class indices that matched the ground truth labels indicating that there were many false negatives.

The top left picture in Figure 16 shows one detection that not only has the wrong label, but also has the wrong bounding box (as the gold items are the only objects in the data set that were not annotated because they are not a type of fruit nut). The remaining detections in the other images in this figure had relatively good bounding boxes, but the class labels were wrong. In the two pictures on the right, a fig was identified as a date and in the picture in the lower left-hand corner, a date was identified as a hazelnut. Interestingly enough, the model identified the gold objects in the top left corner and the date in the bottom left as both belonging to the class hazelnut even though they look nothing alike.

Figure 17 shows predictions made by the model when we do not restrict the class label index to be one of the labels in the ground truth (0, 1, 2). In this case, there are many more object detections made. The numbers in the top left corner of each bounding box represent the predicted class label index. These indices could not be converted to a real label since our training set truth labels only had indices 0, 1, and 2. Some of the bounding boxes are relatively good, but others, such as in the image to the right, take up almost the entire frame or identify one of the golden items that it should not have. The fact that at least some correct bounding boxes were found, but had invalid class index labels, further suggests that the model was unable to learn the different classifications of objects it identified. More evidence of this is in the picture to the left in which some objects were identified multiple times, but with different class labels as can be seen by the overlapping bounding boxes.

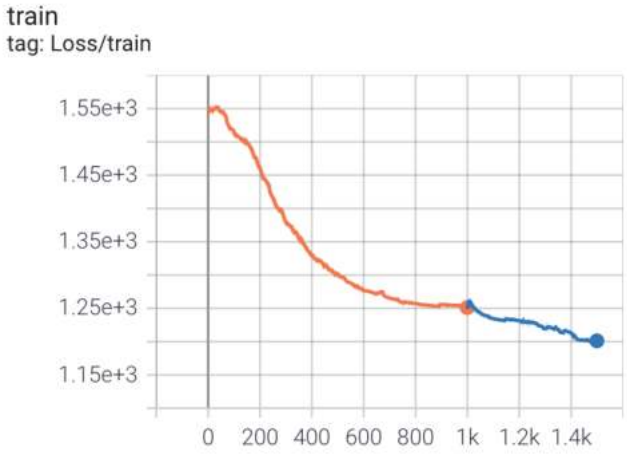

Figure 14: The training loss as epochs elapsed when we trained on the fruit nuts data set. The two different colors represent different tensorboard records. The orange color was when we first started logging. The blue color is the second time we started training (because we were not able to finish training in the time Discovery allows).

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.000
```

Figure 15: Average Precison and Average Recall results from a model we trained on the fruit nuts data set.

Figure 16: Visualization of bounding box detections on the nuts data set from the model we trained after classes have been filtered to range from classification indices 0-2 (the only three indices that were in our truth labels).

Figure 17: Visualization of bounding box detections on the nuts data set from the model we trained when there is no restrictions on class label indices. The numbers in the top left corner of each bounding box represent the predicted class label index.

4.3 Results Using Pre-trained Weights

The pre-trained weights we used were taken from the author of YOLO v3's website [27]. They were trained using the author's own Darknet neural network framework on the COCO train dataset, using multi-scale training, data augmentation and batch normalization [29]. Further details about the pre-trained weights are not provided in either the paper [29] or on the author's website [27]. We loaded these weights into our model and then performed inference using the model.

The results of how our model performed when using pre-trained weights can be seen in Figure 18 and a comparison with other models can be seen in Table 6. Our model underperformed both YOLO v3 and all other object detectors for which we have data, on all metrics except for AP_{75} and AP_M where we outperformed YOLO v2 by just 2.5% on each, and AP_S, where we outperformed YOLO v2 by 4%. In fact, overall, our model performs closest to YOLO v2, only differing within a range of $1.3\% - 8.7\%$ and we approach the AP_S of SSD513 by 1.2%. We also see differences between metrics similar to other models. For example, our AP_{50} is 15% higher than our AP, whereas there is an average 21.2% difference between AP and AP_{50} across the other models. The relative ordering of the precision's magnitudes, from greatest to least of $AP_{50} > AP_L > AP_M > AP_{75} > AP > AP_S$, which is the same across all the different models, is also matched by our model.

On a category by category basis, we can see from Table 7 that the metrics for the dog class consistently outperform all other classes except for the person class which does slightly better in AP_L, AR_M and AR_L. The lowest-performing classes are book and table where the book is generally worse, except for a tie in AP_S and a slightly better performance on AP_M, AR_S, and AR_M.

In performing inference, our model was able to reach a speed of 12.07 fps. This experiment was done by averaging the inference time across 1,000 images on a p100 GPU with 500 GB of RAM. This is significantly slower than the 70-125 fps that YOLO v3 reported and the 65-125 fps that YOLO v4 reported. However,

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.203
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.353
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.217
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.090
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.249
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.318
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.202
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.250
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.250
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.104
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.296
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.398
```

Figure 18: Average Precision and Recall results on the COCO data set when using pre-trained weights.

	backbone	**AP**	$\boldsymbol{AP_{50}}$	$\boldsymbol{AP_{75}}$	$\boldsymbol{AP_S}$	$\boldsymbol{AP_M}$	$\boldsymbol{AP_L}$
Two-stage methods							
Faster R-CNN+++	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI	Inception-ResNet-v2	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	52.1
One-stage method							
Yolov2	DarkNet-19	21.6	44.0	19.2	5.0	22.4	35.5
SSD513	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet	ResNeXt-101-FPN	40.8	61.1	44.1	24.1	44.2	51.2
Yolov3 608 X 608	DarkNet-53	33.0	57.9	34.4	18.3	35.4	41.9
Yolov3 (our model)	**(our model)**	**20.3**	**35.3**	**21.7**	**9.0**	**24.9**	**31.8**

Table 6: Model performance comparison where the evaluation metrics from our model are the results obtained from using pre-trained weights.

our implementation is comparable in speed to other object detection models such as CenterMask (11-15 fps), ATSS (11-19 fps), and EfficientDet (11 - 65 fps) [3].

4.4 Results on Subset of COCO

We trained on a 1,000 image subset of the COCO data set (denoted "Subset of Reduced COCO" in Tables 1 - 3) for a total of 1,000 epochs. A plot of this learning curve can be seen in the left-side image of Figure 21. Data points for this curve were logged in Tensorboard every epoch [23]. For epochs 1-116 we used a learning rate of 1e-5 (the orange line in the picture) before realizing that this was much too high because of the degree to which the loss was fluctuating, so for epochs 116-375 we used a learning rate of 1e-6 (the dark blue and red line in the graph). The loss did not decrease much during these epochs so from epoch 375-433 we tried increasing this to 5e-6 (the light blue line in the graph). Again, we found this was too high (because the loss was greatly fluctuating again) so we dropped it down to 1e-6 (epochs 433 - 615 represented by the magenta color in the graph) and finally down again to 5e-7 (epochs 615 - 1000 represented by the green

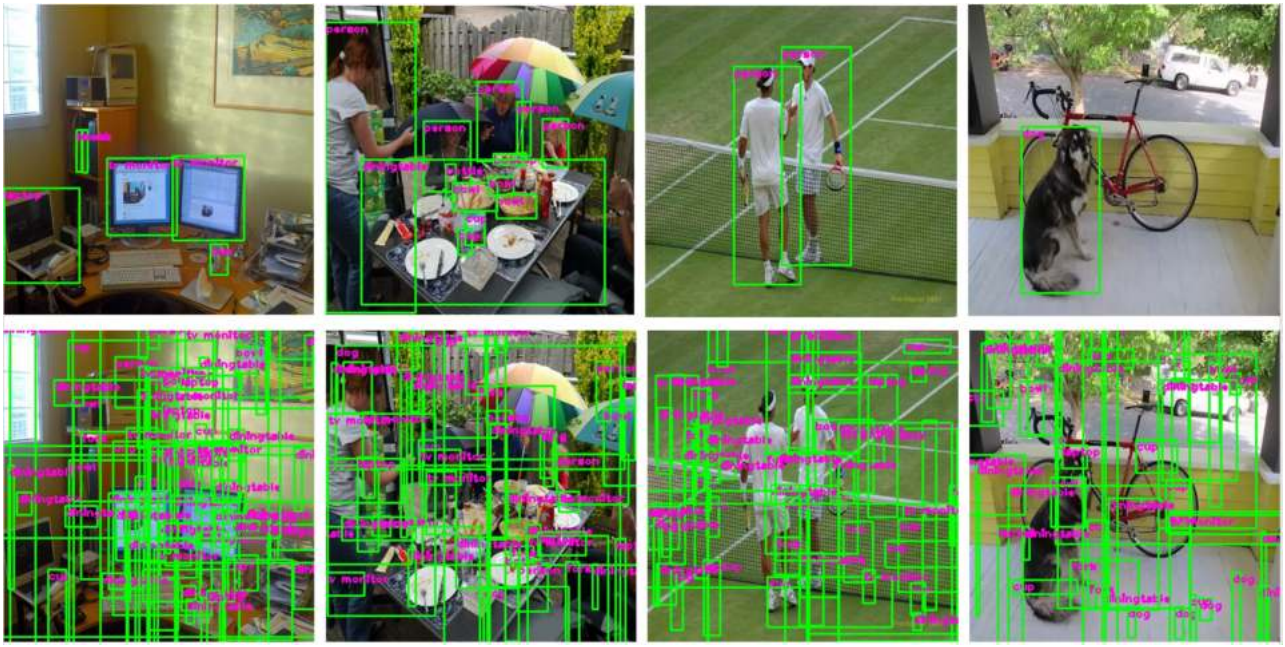

Figure 19: Visualization of predictions from a YOLO v3 model. The first row are detections obtained using pre-trained weights and the bottom row are the same images but with detections from the model we trained. Both detection outputs have been restricted to only consider the 10 categories we specified in our reduced data set.

Category	**AP**	AP_{50}	AP_{75}	AP_S	AP_M	AP_L	**AR**	AR_S	AR_M	AR_L
Book	0.016	0.038	0.011	0.010	0.047	0.037	0.025	0.013	0.079	0.066
Bottle	0.123	0.251	0.096	0.071	0.252	0.351	0.168	0.098	0.329	0.425
Bowl	0.161	0.290	0.162	0.072	0.225	0.239	0.228	0.099	0.296	0.354
Cup	0.169	0.291	0.180	0.081	0.279	0.424	0.217	0.109	0.343	0.506
Dog	0.415	0.653	0.471	0.268	0.397	0.468	0.487	0.289	0.437	0.566
Fork	0.101	0.241	0.090	0.053	0.199	0.129	0.127	0.049	0.251	0.214
Laptop	0.317	0.504	0.368	0.114	0.332	0.392	0.361	0.118	0.356	0.458
Person	0.279	0.516	0.276	0.123	0.369	0.505	0.336	0.153	0.440	0.604
Table	0.120	0.218	0.119	0.010	0.045	0.201	0.168	0.011	0.049	0.291
TV	0.330	0.524	0.394	0.100	0.340	0.432	0.383	0.100	0.384	0.500

Table 7: Average Precision and Average Recall per category when using pre-trained weights with our model.

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.001
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.002
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.002
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.002
```

Figure 20: Average Precision and Average Recall results for the model we trained ourselves and evaluated on the COCO data set.

and gray lines in the graph). Every 20 epochs we also logged the validation/test loss which can be seen in the right-side image of Figure 21. The lowest loss we achieved on the training set was approximately 9,120 which occurred around epoch 200 and for the test set the lowest was around 9,600 (also around epoch 200).

The results of our trained model can be seen in Figure 20. These results are after the predictions have been filtered from the 80 possible class labels the model can predict down to the 10 that we actually trained on. Most of the average precision and recall values were zero, implying that the model performs very poorly. This also indicates that the number of true positives is very low. The only values in the figure that are not zero are those for AR_{all} for both maximum detections equal to 10 and 100, and for AR_M and AR_L. The highest Average Recall percentage was only 0.2%.

Visualization of bounding box predictions using our trained model for our 10 classes can be seen in the bottom row of Figure 19. It is evident that the pre-trained model (whose predictions are the top row of Figure 19) performs much better for these four images as it has not only correctly classified objects but also does not have any false positives. Our trained model, on the other hand, has upwards of 30 overlapping bounding boxes in each image. It misclassifies every (or nearly every) part of each of the four images which imply that the model has not been able to learn the true characteristics of each object. As can be seen in the figure, there is a large number of false positives.

5 Conclusion

5.1 Discussion of Results

5.1.1 Discussion of Pre-trained Weights Performance

It is evident that objects that are smaller in size such as book, bottle, bowl, cup, and fork, are harder to predict and result in lower precision compared to larger objects like person, TV, laptop, and dog. This is consistent both with the observation made in YOLO v3 that "in the past YOLO struggled with small objects" [29], and the data presented in Table 6 in which AP_S is the lowest-performing metric across all models for which we have data, whereas AP_L is the second highest, indicating that detection of smaller objects is generally a problem in the field of object detection.

For books in particular, which was one of the lowest-performing categories we tested, we think that the positioning of the object may play a big role. For instance, books can be open or closed in different images which makes them look different and may explain the result of the model having trouble identifying them.

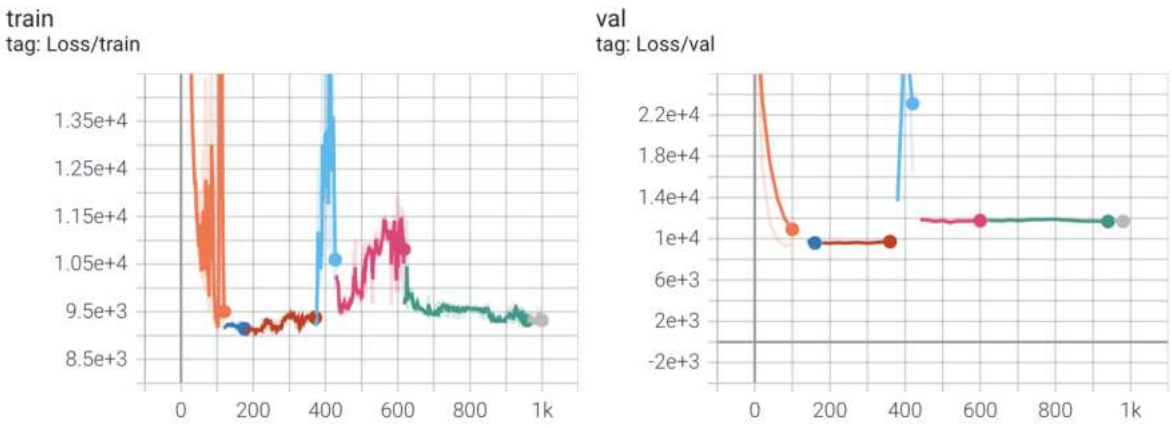

Figure 21: The training loss (left) and validation loss (right) as epochs progressed when we trained on the subset of the COCO data set. The different colors indicate different runs of the training program. This was necessary when we manually changed the learning rate or when we maxed out on time for Discovery.

We also noticed that in many of the COCO images that the pre-trained weights were trained on, the books were in the background and hard for even a human to recognize without taking context into consideration. Additionally, there were many images of books on bookshelves in which only a handful of the books had annotations. Not having accurate labels likely impacted the training of the model on this category and thus impacted the weights we loaded.

Overall, dog is the best performing category, possibly because of how large and distinctive they usually are. Another reason that it may have been easier for our model to detect dogs is because we do not have any other animal or category that closely resembles a dog in our data set. Despite these factors, it is still somewhat of a surprise given that the dog class has some of the fewest images and annotations of the classes in the COCO data set.

While the person class had the greatest number of images and annotations, it lagged behind the dog class in almost all metrics except for AP_L, AR_M and AR_L. One possible explanation for this, as with some other classes, is the natural variety that exists within the class. We may expect great diversity in the types of clothing that people are wearing, the environments they are surrounded by, and the activities which they are taking part in. However, the rest of our results seem consistent with the fact that person had the most annotations in the training set because although it did not always outperform dog, it was still one of the better performing categories.

A surprise was the relatively low performance on the dining table class. Given the generally large size of dining tables and the number of images and annotations of them in the COCO data set, we might have expected a higher average precision. A possible reason for this low precision is that dining tables in the images are defined very broadly with a high variety of shapes and sizes. For example, some were rectangular and large and others were small and round. In addition, in many images tables are covered with several other objects, which may have made them harder to detect.

Overall, our general underperformance is probably what surprised us the most, both since we located and used the original YOLO v3 weights file, and because visual inspection of the detections made by our model using the pre-trained weights looked accurate (see Figure 19). Since we would expect similar results to those obtained by YOLO v3 based on these facts, we are not entirely sure what explains our model's

relatively poor performance, and will need to further investigate to determine if there is a bug somewhere, perhaps in our evaluation pipeline or somewhere deep in the network architecture.

One thing that did make sense to us was the frame rate we achieved. While we were hoping for something higher than 12.07 fps, it makes sense that we are considerably slower than what YOLO v3 reported. This is because all of our code was done in Python, which is a very slow language, and our code was not optimized. YOLO v3 on the other hand was implemented in C. If we wanted to get similar results in the future, we would have to probably use a combination of Python and C++ or use C like YOLO v3.

5.1.2 Discussion of Models Trained by Us

As can be seen in Sections 4.2 and 4.4, the models we trained performed extremely poorly. Our model was unable to learn anything useful and had very few true positives and for at least the COCO data set, many false positives.

For the nuts data set, only a handful of detections were made when the class label indices were filtered, some of which were false positives and others that had decent bounding boxes but wrong class labels. When we lifted the restrictions on class labels, we got more detections, but all of the class labels are wrong. An indicator that something was wrong is the learning curve (Figure 14). The loss even after 1500 epochs is extremely high– around 1200. One possible reason is that we used too low of a learning rate for the first 1000 epochs.

For the COCO data set, we had large numbers of false positives and many overlapping bounding boxes. Like the nuts data set, it can be seen in the learning curve (Figure 21) that we had problems with the learning rate. The loss either fluctuates greatly or steadily starts increasing at different sections. This is definitely one area that negatively impacted our results. Like the training on the nuts data set, we had very high losses even during later epochs.

The large losses we had while training both models, as well as the poor performance of the models, indicate that there is something wrong with our set-up. We are not sure if the error pertains to the model, the loss function, the training loop, or possibly some combination. More discussion about these possible factors can be found in the following section.

There are a few obvious things we noticed that went wrong though. For one, training (especially for the nut data set) was most likely negatively impacted by the fact that we allowed our model to make 80 different class predictions, but both of our data sets had fewer than 80 classes. This may be one reason to explain why all (or almost all) of the class labels were incorrect. Additionally, the excessive number of overlapping bounding boxes in the bottom row of Figure 19 indicates that our non-maximum suppression (NMS) may not be working. It is hard to tell for sure because our NMS function only takes into account overlapping bounding boxes of the same label rather than different labels. It's difficult to see by simply analyzing the pictures if any of the overlapping bounding boxes are of the same label. The last observation brings us to the other area we went wrong– having overlapping bounding boxes of different labels (such as the left image in Figure 16). In the future, we would detect overlapping bounding boxes of different labels and only keep the one that had the highest confidence score. It's possible the latter two problems would not happen if the model had been successfully trained.

One thing that we are unable to explain is why the training we did on the COCO data set resulted in over 20 bounding boxes per image (at least for the images we visualized results on), but the nuts data set did not have this problem. Many of the bounding boxes for the nuts data set were wrong, but there weren't nearly as many false positives as in the model we trained on COCO. This seems weird because the code and

training process was identical for these two models with the exception of the learning rates used. A potential reason for this is that the nuts data set contains fewer classes with just three categories, compared to the ten categories in our reduced COCO data set.

5.2 Possible Reasons for Inability to Train

We had a lot of issues training and were not able to achieve low losses even after many epochs had elapsed. Initially, this was due to our computation graph getting disconnected and because of errors in our loss function which we had to spend a great deal of time debugging. We noticed there was something wrong with our model because backpropogation was not working. Even after having the optimizer take a step, the gradients were not changing and as a result the loss never changed. To identify the problem, we utilized the library called pytorchviz which renders a diagram of the computation graph [37]. We found that the computation graph was getting disconnected in the very last layer. After some troubleshooting, we discovered that we had written the detection layer of the architecture as an additional utility function when it actually needed to be the forward function of an nn.Module subclass. The visualization produced by pytorchviz, once we had fixed this error, is included in the submission zip file as supplementary material, titled 'pytorchviz.pdf'.

In regards to our loss function, there were several things we checked. One thing we checked was whether our targets were in the correct format. This was necessary because the bounding boxes in the JSON file that had our truth labels had a different format than the predictions tensor our model outputted. We checked this by saving the truth tensor generated in our loss function and then using it to draw bounding boxes on the original image. We also were unsure whether the mean square error calculations using Pytorch's built-in MSE class were working as expected so we recorded all of the truth and predicted bounding boxes for one image and did the calculations by hand to make sure they matched. Lastly, we stepped through our loss function with a debugger in an IDE and made sure that everything was behaving as expected.

While debugging, we did find a few notable mistakes. The first mistake we made was due to a miscommunication between teammates. The person who wrote the loss function was different than the one who wrote the model architecture and had an incomplete understanding of the format of the predictions tensor. Initially, the loss function expected the predictions tensor to have bounding box (x, y) coordinates relative to their offset position in the grid cell. However, the way the model was constructed, the predictions give the absolute (x, y) coordinates in the image. We were alerted to another mistake when we realized that the magnitude of the different components of MSE losses were several orders of magnitude apart. After re-reading the YOLO v1 paper, we realized we were not normalizing the bounding box coordinates and width and heights. This resulted in extremely high losses (in the millions) because the bounding box errors were so large in magnitude and then were being multiplied by 5 in addition to that because that's how the original paper presented it. Because the bounding box losses were so high, backpropogation seemed to focus more on these errors than any of the other loss components in Figure 7. As a result, our initial trials at training had especially bad results in the classification aspect of the detection. Once we normalized the bounding box truth and prediction tensors to lie between 0 and 1, our losses were then only in the thousands and we had a little more success in training, especially in regards to the resulting classifications. While we still couldn't classify accurately, we at least had more classifications of the valid class options. Another thing we changed after adding normalization was to make sure that when we normalized the bounding boxes, none of those computations impacted the gradients during backpropogation. We did this by enclosing the normalization code in "torch.no_grad()". After we made these changes, we did a sanity check by using our loss function to compute the loss between good predictions on one image with a model loaded with pre-trained weights and

the true labels and got a loss of 24. Since the loss was low and seemed reasonable, we believe that our loss function is probably correct.

However, even after fixing these mistakes, we were unable to get very low losses during training indicating that there are probably other issues we have not identified. One issue we had was picking hyper-parameters such as learning rate. We were not able to find good values for this. We started off with what the YOLO v2 paper suggested, which ranged from 10^{-2} to 10^{-3} to start and then decreased as time went on, but we found this was much too high. When we plotted the loss curves, the curve was erratic and went up and down by large amounts. We kept slowly lowering it until we saw that it was consistently going down. However, as time went on, we found that the loss leveled out and was not decreasing so in between epochs we again played with changing the learning rate. We waited several epochs, looked at the curve again, and then decided whether to continue adjusting the rate. In the future, we would have to come up with a better system than this and perhaps write a custom learning rate scheduler to meet our needs and also add gradient clipping.

Another issue we found, which has been mentioned previously, was that because of the way our model was set up, it was expecting there to be 80 classes when for our data sets we only had 3 or 10 classes. This gave it the opportunity to make wrong predictions because it could predict classes that did not exist in our data set. It is very possible that this negatively impacted training.

Other than adjusting the learning rate and fixing the model to be more flexible with the number of classes it can predict, it's not clear where else we went wrong. The model performed decently with pre-trained weights indicating that the model architecture is likely correct, or at least capable of functioning well. It's still possible there is a bug in the architecture we did not find though since it did perform so much worse than what the YOLO v3 paper reported. The loss is low when evaluated on images with pre-trained weights indicating that the loss function may be correct. The structure of the training loop is nearly identical to tutorials on Pytorch's website so there is little room for error there. Much more investigation is needed to determine what exactly is wrong with our set up.

5.3 Other Challenges We Faced and Things that Went Wrong

One of the first major setbacks we encountered was that after we had written the majority of our code in Jupyter notebooks on our group's Google Colab we discovered that the upload rate to transfer our data to the Google drive was prohibitively slow. While we experimented with potential workarounds, we also tried training our model on a smaller, 18 image dataset, only to discover that our notebook would quickly run out of RAM. Due to these problems, it became clear that we would need to transfer to a Discovery cluster.

To do this we first needed to convert our Jupyter notebook files into stand-alone python scripts that could be run from a command-line interface. We then used SCP to transfer our data to Discovery and cloned our Gitub repo to access our Python scripts.

Since Discovery is accessed via ssh through a terminal it required some experimentation to figure out how to correctly use it. We initially attempted to use a Tmux session to run our code which we would detach from to leave our job running while offline. However, this technique turned out to be infeasible when combined with requesting a node with the compute resources to run our code. We also initially had some trouble figuring out how to utilize the CUDA GPUs on Discovery to speed up our training so our first attempts at training were extremely slow. PyTorch requires that every time a new tensor is instantiated it is loaded onto the GPU using a tensor.cuda() method. However, for the code to also work on a regular CPU, these method calls have to put inside torch.cuda.is_available() conditionals. After some troubleshooting, we located all such instances and immediately noticed a huge speed improvement.

Although the speedup was significant, it was not enough to complete sufficient training epochs within our eight-hour time limit. As mentioned earlier, our solution was to use PyTorch's checkpointing system to save the current state every five batches. To make the best use of our time, team members took 'shifts' with some members starting training late at night and letting another member restart training first thing in the morning.

There were two other main challenges we faced. The first was the relative inexperience of team members to machine learning and deep learning. This course was the first machine learning experience all team members had and none of us had used Pytorch previously. We learned a lot while troubleshooting at office hours, but we really lacked a lot of deep learning knowledge that was needed to do this project. The second challenge was the time constraint. We found that when combined with our other coursework, this project was too much to complete in six weeks.

5.4 Future Work

If we had more time, there are several things we would do to improve this project. The first would be to make the model more flexible. We would like to change the model so that you could specify any number of classes and not have to use the number 80.

The biggest thing we would like to fix is our ability to train. We got decent results with the pre-trained weights, but very poor results when we attempted to use our loss function to train ourselves. Although as mentioned previously, the loss function and architecture may in fact be correct, there is something wrong with our set-up that needs further debugging since we were never able to attain low losses even after many epochs. Part of this may be solved by better adjusting hyperparameters like learning rate, but it not apparent if that alone would fix it. It would be very satisfying to fix this and be able to play more with different hyperparameters and settings to see if we could improve our results.

Once the training loop/loss function was fixed we would like to continue with our original goal of training on the augmented data set we created. It would be interesting to do this and then compare the results we would be able to get from using pre-trained weights, training on the subset of COCO we created, and training on the augmented data set and see which performs best. We could also play with things while doing this like changing the anchor box priors to those we found using k-means versus using the priors the paper used. Based on these results, if we find that the augmented data performed worse, then this may indicate that there is additional future work in modifying the augmentation pipeline to overcome some of the problems discussed in Section 4.1. One modification we can think of already for the augmentation pipeline is transferring the code from Python to C++ (a language that also has an OpenCV library) in order to increase speed. We could also transfer some of our model Pytorch code to C++ to allow for faster inference speeds.

Lastly, the other obvious area of improvement that we could make is to add in features of YOLO v4 to our project (which currently is mainly based on YOLO v3). The only current feature of YOLO v4 that we have is some of the data augmentation techniques such as cutmix and mosaic. YOLO v4 uses a slightly different backbone in their architecture than what we used, called CSPDarknet53, and also adds DropBlock regularization, class label smoothing, and different activations (such as Mish activation), among other things. Perhaps the biggest change between YOLO v3 and v4 that we could tackle is the new loss function: CIoU-loss. CIoU-loss (Complete Intersection over Union) is a variant of IOU (Intersection of Union) loss that considers overlapping areas, aspect ratios, as well as the distance between center points, which helps in 'normalizing' the loss across different scales.

Although there is still future work to be done, we are happy with the amount we accomplished in six weeks, especially given the advanced nature of object detection and the relative inexperience of the team.

References

[1] *Albumentations Documentation*. (Accessed on 12/15/2020). DOI: {http://albumentations.ai/docs/}.

[2] *An Introduction to Evaluation Metrics for Object Detection*. (Accessed on 10/18/2020). DOI: {https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/}.

[3] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: *arXiv preprint arXiv:2004.10934* (2020).

[4] Rung-Ching Chen et al. "Automatic License Plate Recognition via sliding-window darknet-YOLO deep learning". In: *Image and Vision Computing* 87 (2019), pp. 47–56.

[5] *COCO - Common Objects in Context Download*. (Accessed on 10/17/2020). DOI: {https://cocodataset.org/#download}.

[6] *COCO - Common Objects in Context Metrics*. (Accessed on 10/18/2020). DOI: {https://cocodataset.org/#detection-eval}.

[7] *cocodataset/cocoapi: COCO API - Dataset @ http://cocodataset.org/*. (Accessed on 10/17/2020). DOI: {https://github.com/cocodataset/cocoapi}.

[8] Terrance DeVries and Graham W Taylor. "Improved regularization of convolutional neural networks with cutout". In: *arXiv preprint arXiv:1708.04552* (2017).

[9] *Evaluating Object Detection Models: Guide to Performance Metrics — Manal El Aidouni*. (Accessed on 10/18/2020). DOI: {https://manalelaidouni.github.io/manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html}.

[10] *Evaluation metrics for object detection and segmentation: mAP*. (Accessed on 10/18/2020). DOI: {https://kharshit.github.io/blog/2019/09/20/evaluation-metrics-for-object-detection-and-segmentation}.

[11] *Fruits Nuts Segmentation Data Set*. (Accessed on 12/14/2020). DOI: {https://github.com/Tony607/mmdetection_instance_segmentation_demo}.

[12] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[13] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[14] Kaiming He et al. "Mask RCNN. arXiv e-prints, Article". In: *arXiv preprint arXiv:1703.06870* (2017).

[15] Miao Kang et al. "Contextual region-based convolutional neural network with multilayer fusion for SAR ship detection". In: *Remote Sensing* 9.8 (2017), p. 860.

[16] Ayoosh Kathuria. *How to implement a YOLO (v3) object detector from scratch in PyTorch*. (Accessed on 12/12/2020). DOI: {https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/}.

[17] Ayoosh Kathuria. *What's new in YOLO v3?* (Accessed on 12/12/2020). DOI: {https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b}.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[19] Wenbo Lan et al. "Pedestrian detection based on yolo network model". In: *2018 IEEE international conference on mechatronics and automation (ICMA)*. IEEE. 2018, pp. 1547–1551.

[20] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in network". In: *arXiv preprint arXiv:1312.4400* (2013).

[21] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[22] Li Liu et al. "Deep learning for generic object detection: A survey". In: *International journal of computer vision* 128.2 (2020), pp. 261–318.

[23] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[24] Mohammed A Al-Masni et al. "Simultaneous detection and classification of breast masses in digital mammograms via a deep learning YOLO-based CAD system". In: *Computer methods and programs in biomedicine* 157 (2018), pp. 85–94.

[25] Aladdin Persson. *Pytorch YOLO From Scratch*. (Accessed on 12/12/2020). DOI: {https://www.youtube.com/watch?v=n9_XyCGr-MI}.

[26] Sovit Ranjan Rath. *Evaluation Metrics for Object Detection*. (Accessed on 10/18/2020). DOI: {https://debuggercafe.com/evaluation-metrics-for-object-detection/}.

[27] Joseph Redmon. *YOLO: Real-Time Object Detection*. (Accessed on 12/14/2020). DOI: {https://pjreddie.com/darknet/yolo/}.

[28] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[29] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[30] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[31] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *IEEE transactions on pattern analysis and machine intelligence* 39.6 (2016), pp. 1137–1149.

[32] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[33] Pierre Sermanet et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks". In: *arXiv preprint arXiv:1312.6229* (2013).

[34] Connor Shorten and Taghi M Khoshgoftaar. “A survey on image data augmentation for deep learning”. In: *Journal of Big Data* 6.1 (2019), p. 60.

[35] Cheng Wei. *How to train Detectron2 with Custom COCO Datasets*. (Accessed on 12/15/2020). DOI: {https://www.dlology.com/blog/how-to-train-detectron2-with-custom-coco-datasets/}.

[36] Sangdoo Yun et al. “Cutmix: Regularization strategy to train strong classifiers with localizable features”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 6023–6032.

[37] Sergey Zagoruyko. *Pytorchviz: A small package to create vizualizations of PyTorch execution graphs*. (Accessed on 12/15/2020). DOI: {https://github.com/szagoruyko/pytorchviz}.