OBEJECT DETECTOR IMPLEMENT VIA YOLO V3 ALGORITHM

Zihao Zhu
Student ID: 52304381
zzhu20@uci.edu

Perry Liu
Student ID: 68325714
peiyanl3@uci.edu

ABSTRACT

We are going to build an object detector based on YOLO v3 algorithm. YOLO stands for You Only Look Once. It's an algorithm for object detector that uses features learned by a deep convolutional neural network. This algorithm is extremely fast and accurate. Our group will build it from scratch with the help of the paper "YOLOv3: An Incremental Improvement".

1 INTRODUCTION

Objection detection is one of fundamental problems of computer vision and has a plenty of related application, such autonomous robot, self-driving techs, and etc. With modern machine learning techniques, people are trying to design numerous object detection algorithms to improve the speed and accuracy. Recent years have seen people develop many algorithms for object detection, some of which include YOLO, RetinaNet, Mask RCNN and SSD. In this project, we will build an object detector with the YOLO v3 algorithm with Pytorch and explore the aptitude of the application.

The purpose of this project is to detect various objects in a single image with our model. Some of the objects may overlap or only have portions of features shown in the image. The goal is to output the right prediction even with the influence of those obstacles.

Unlike other object detection algorithms, YOLO only uses convolutional layers, which makes it become a fully convolutional network. It does not use any form of pooling. Instead, it utilizes a convolutional layer with stride 2 to downsample the feature map. In light of that, it effectively helps prevent the loss of low-level features that are often caused by pooling. For inputs, YOLO processes the image by splitting it into grids of size n by n cells, and each cell will be responsible for predicting different things. For outputs, the prediction is done by using a convolutional layer that uses 1 x 1 convolutions. The important thing about the output is that it's actually a feature map. We will introduce it details in the later section.

Another important concept is anchor boxes. Anchor boxes are a set of predefined bounding boxes of a certain height and width. It allows one grid cell to detect multiple objects. YOLO v3 has three anchors, which result in the prediction of three bounding boxes per cell.

YOLO can only detect objects belonging to the classes present in the official COCO dataset, and we will use pre-trained official weights file for our detector. The weights are obtained by training the algorithm on COCO dataset, and therefore we can have 80 object detection categories.

2 RELATED WORKS

As an object detection algorithm, YOLO v3 can be used extensively in the implementation of various applications. Since YOLO v3 provides a relatively quick training time and a decent accuracy with a large enough dataset, it delivers optimal performance when compared to other algorithms like RetinaNet. With its applicability, the YOLO v3 algorithm is utilized in many applications in fields such as computer vision and image processing. Some common examples that we see in our day-to-day lives include general object detection, traffic flow detection, and autonomous driving.

Applications that identifies objects in an image or a video are quite common in our daily lives. For instance, the medical industry uses object detectors frequently to utilize the power of deep learning. In medical practice, an abundant amount of medical images are generated each day and are examined by medical professionals. This process is time-consuming and sometimes error-prone. In this scenario, object detectors, specifically ones that detect tumors, polyps, and diseases, are trained and applied to identify the respective special objects in the medical images. An example of a portable imaging system has shown that YOLO v3 can be used to identify brain tumor with decent accuracy and speed [1]. This largely streamlines the diagnosis process, which leads to the medical professionals being able to treat patients more efficiently.

Traffic flow detection and autonomous driving are also technologies that implement object detection. Both traffic flow detection and autonomous driving solve the same problem of detecting potentially high-speed objects on the road. In this type of detection applications, the algorithm has to be quick and accurate as the detection is often applied on a live feed rather than a still image. Therefore, this scenario incentivized the balance between accuracy and runtime for the algorithm. Researches have indicated that variations of the YOLO v3 algorithm can support a good balance of accuracy and runtime in the problem of detecting moving objects, potentially at a high speed, in a live feed [2][3].

As mentioned previously, YOLO v3 is an algorithm widely applied in the field of computer vision. Utilizing a fully convolutional network, YOLO v3 is able to detect objects in real time. Like its previous iterations, YOLO v3 can be applied to images, videos, and even live feeds, its applications include identifying vehicles in traffic and live feed identification. Some related algorithms and applications include RetinaNet and Single Shot Multi-Box Detector (SSD).

Similar to YOLO v3, RetinaNet utilizes convolutional neural networks, downsampling, and upsampling. However, the RetinaNet model tend to use ResNet as the backbone of the algorithm. The architecture of RetinaNet also includes a feature pyramid network which is responsible for the classificaitions. In comparison, the architecture of YOLO v3 classifies the image in sequential layers [4].

The Single Shot Multi-Box Detector(SSD) is also largely similar to YOLO and RetinaNet, but uses ground truth information in regards to certain outputs. This special implementation detail of simplifies the learning process in the layers, which results in relatively quicker training time [4].

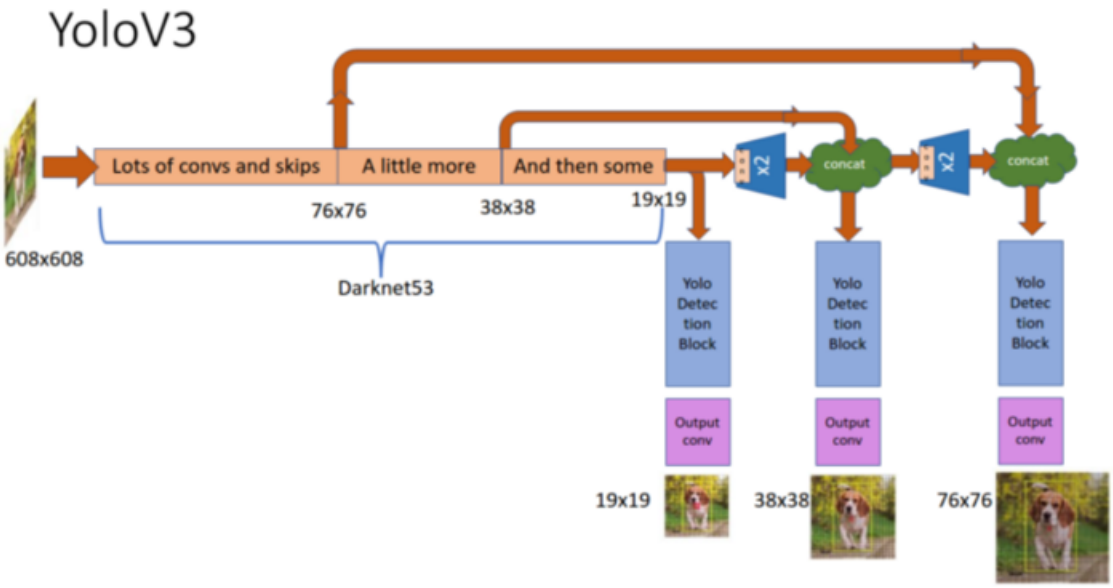
Comparing these models, RetinaNet can achieve high accuracy with higher training time, while SSD can produce a relatively accurate result in a short time [4]. However, the updates in YOLO v3, compared to previous iterations, have been observed to outperform these other algorithms.

3 DATASET

In our project, we explored the performance of the YOLO v3 algorithm with the same dataset, COCO (Common Objects in Context), as the one used in Redmon's paper. COCO is a dataset from Microsoft that is comprised of more than 300,000 images and more than 200,000 of them are labeled. These images include photos comprised of objects from 80 categories: person, bicycle, car, motorcycle, etc. The objects are in photos of everyday scenes, which places them under their natural context. Since the object images are labeled, this dataset, and any subset from it, is commonly used in training convolutional neural networks that targets the object detection problem. With inputs from the COCO dataset, our convolutional neural network, which implemented the YOLO v3 algorithm using DarkNet, will be able to learn how to identify objects that occur in natural contexts. The context in which the objects are placed in is quite significant; as the convolutional neural network train on the images, different contexts of objects can be helpful to isolate the specific features of the objects, which will then improve the performance of the model. In our model, we imported pretrained weights from the tutorial[6], which is also trained on the COCO dataset.

4 METHOD

The overall architecture of YOLO v3 looks like this:



Darknet53 is the fully convolutional neural network. To build the network, we use a configuration file provided by the authors of YOLO v3. There are different blocks in the configuration file: convolutional, shortcut, upsample, route, YOLO, and net.

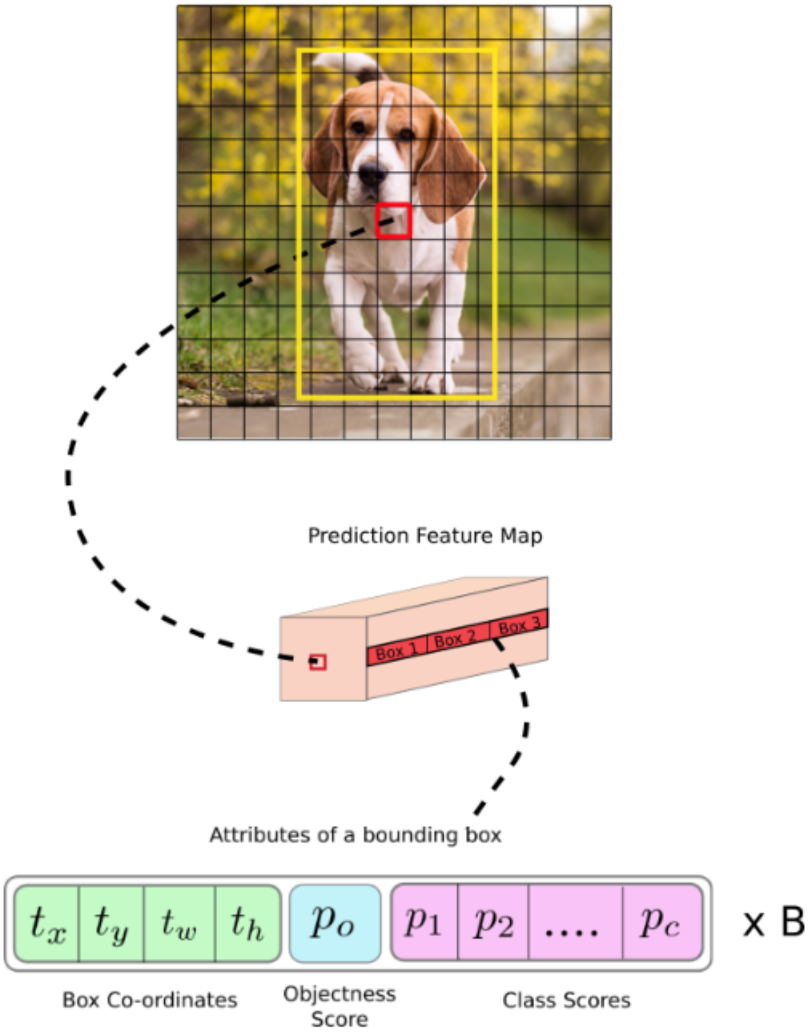
Convolutional layers are exactly what we saw in the CNN. In each of those layers, there are attributes: **batch_normalize**, **filters**, **size**, **stride**, **pad**, and **activation**. All of those are predefined, so we don't have to fine-tune them. Shortcut layers play a role of skip connection, which is akin to the one used in Resnet. There is a **from** attribute. For instance, if we set from = -2, it means the output of the shortcut layer is obtained by adding feature maps from the previous and the 2nd layer backwards from the shortcut layer. Upsample layers will upsample the feature map in the previous layer by a factor of **stride**, an attribute, using bilinear upsampling. Route layers have an attribute **layers**, which can have 1 or 2 values. When layers attribute has only one value, it outputs the feature maps of the layer indexed by the value. When layers has two values, it returns the concatenated feature maps of the layers indexed by it's values. YOLO layer corresponds to the Detection layer in the following part. At here, it defines 9 predefined anchors but indicates we would only use 3 of them. It corresponds to the mechanism that each cell of the detection layer predicts 3 boxes. The net layer describes information about the network input and training parameters. It isn't used in the forward pass of YOLO. However, it does provide us with information like the network input size, which we use to adjust anchors in the forward pass.

In our code, we use a **create_modules** function to take those predefined blocks and return a **nn.Modulelist**. In this function, we iterate over the list of blocks and create **nn.Module** object for each block. After this, we will have all the net information and module lists we need.

Before we implement the Darknet model, we also need a function: **prediction**. It takes an detection feature map and turns it into a 2-D tensor, where each row of the tensor corresponds to attributes of a bounding box. Remember, YOLO v3 algorithm produces a feature map as an output. Since we have used 1 x 1 convolutions, the size of the prediction map is exactly the size of the feature map before it. In other words, each cell can predict a fixed number of bounding boxes. In YOLO v3, each cell will predict 4 coordinates for bounding box, probability that a box contains an object, and class probabilities for each of the class.

Depth-wise, we have $(B(5 + C))$ entries in the feature map. B represents the number of bounding boxes each cell can predict. According to Redmon's paper, each of these B bounding boxes may specialize in detecting a certain kind of object. Each of the bounding boxes have $5 + C$ attributes, which describe the center coordinates, the dimensions, the objectness (bounding box probability) score and C class confidences for each bounding box [5]. As we mentioned earleir, YOLO v3 has three anchors per cell, and therefore, it can predict three bounding boxes for every cell.

It's basic structure looks like this:



The predictions correspond to:

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

where b_x, b_y, b_w, b_h are the x,y center co-ordinates, width and height of our prediction. t_x, t_y, t_w, t_h is the outputs of the network. c_x and c_y are the top-left co-ordinates of the grid. p_w and p_h are the box's anchors dimensions.

The reason why we use a sigmoid function on our center coordinates is because we want to force the outputs to be between 0 and 1. To explain this, we should know that YOLO doesn't predict the absolute coordinates of the bounding box's center. It predicts offsets that are relative to the top left corner of the grid cell which is predicting the object, and Normalised by the dimensions of the cell from the feature map, which is 1. The tutorial shows us a vivid example:
"For example, consider the case of our dog image. If the prediction for center is (0.4, 0.7), then this means that the center lies at (6.4, 6.7) on the 13 x 13 feature map. (Since the top-left co-ordinates of the red cell are (6,6)). But wait, what happens if the predicted x,y co-ordinates are greater than one, say (1.2, 0.7). This means center lies at (7.2, 6.7). Notice the center now lies in cell just right to our red cell, or the 8th cell in the 7th row. This breaks theory behind YOLO because if we postulate that

the red box is responsible for predicting the dog, the center of the dog must lie in the red cell, and not in the one beside it."[7]
Therefore, to avoid this problem, by limiting the range of outputs between 0 and 1, it effectively keeps the center in the grid which is predicting.

The dimensions of the bounding box are predicted by applying a log-space transform to the output and then multiplying with an anchor. From the paper [4], we are told that "The resultant predictions, b_w and b_h, are normalised by the height and width of the anchors.So, if the predictions bx and by for the box containing the dog are (0.7, 0.8), then the actual width and height on if we have anchor box of height 10 and width 8 is is (10 x 0.7, 8 x 0.8)."

Again, YOLO v3 makes prediction across 3 different scales. The detection layer is used make detection at feature maps of three different sizes, with strides 32, 16, 8 respectively. For example, if we have an input with size 416 x 416, the model would make detection on scale 13 x 13, 26 x 26 and 52 x 52. For each scale, there would be 3 corresponding anchors per cell. Hence, the total output would be $((52 * 52) + (26 * 26) + 13 * 13)) * 3 = 10647$ bounding boxes.

With all the preparation above, now, we are able to implement the Darknet model. Since we've already had everything we need, for each block among the six (convolutional, shortcut, upsample, route, YOLO, and net) in the module list, we only have to load corresponding module object and use functions we have defined to finish the forward pass. Particularly, we will use the pre-trained weight from YOLO v3 official. The weights for these layers are stored exactly in the same order as they appear in the configuration file. So, we just load weights into each layer in the same order. At this point, we have finished the implementation of the Darknet-53.

Early, we use an example to illustrate that there would be 10647 bounding boxes for a 416 x 416 image. To reduce the detection from 10647 to 1, we will have to use thresholding by non-max suppression. The goal of non-max suppression is to eliminate the scenario that there are multiple detection box for one object. To do so, suppose we have two lists A and B where A contains all the proposals (information of the prediction on a object) and B is a empty list, the first step is to select the proposal with highest confidence score, and remove it from A and append it to B. Then we should calculate the IOU of this one with every other proposal. In this loop, if the any of IOU is larger than the threshold value, we would remove that proposal from A and append it to B. We will repeat this process until there's nothing left in A.

Finally, we can get a prediction in the form of a tensor.

5 Result and analysis

5.1 evaluation metrics

The evaluation would be divided into two parts: 1. whether the object detector can output a correct detection, and 2. whether the output box can precisely bound to the object. There are 100 test images for evaluation. Some of them only contain one object to detect, and in the meanwhile, some of them contain more than 10 objects to detect. In light of that, we define our test images into three levels: easy, medium, and hard. We will calculate performance scores for both parts of evaluation. We define that easy images have 1 point, medium images have 3 point, and hard images have 6 point. We have 50 easy images, 30 medium images, and 20 hard images in total.

For the first part, we calculate a detection score. The maximum score is $50 + 30 * 3 + 20 * 6 = 260$. The scoring policy for each image is that if all the objects can be correctly detected, it will get full point; if 20% of the objects are incorrectly detected, it will get half of the full points; if more than 50% of the objects are incorrectly detected, it will get 0 point.

For the second part, we use user studies to evaluate whether the detection box can precisely bound to the object. Since there is not a clear method to evaluate how well the detection box bound to the object, we choose to ask humans instead computer to evaluate the performance. We recruited 10 participants who have done computer vision related projects before for our user studies. For each image, they will be asked for a percentage of how well those boxes bound to the objects, and we will take an average from all participants and calculate a corresponding score. For instance, if the test image is a medium image and the average rating is 90%, it will get a score of $3 * 90\% = 2.7$ point.

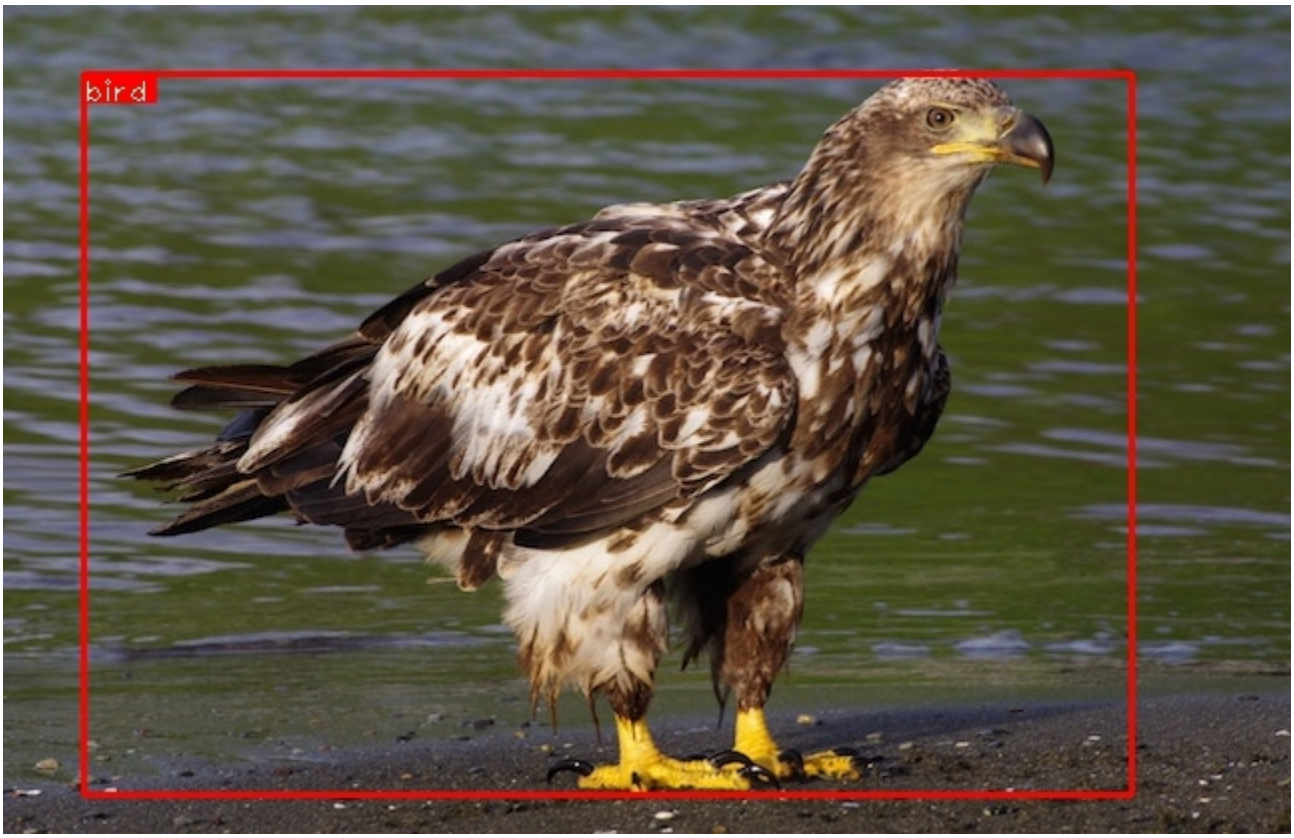
5.2 RESULT

	easy	medium	hard	total
detection score	50	30*3	17*6 + 3*3	251
bounding box score	50	81	99	230

In the detection evaluation part, all the objects in easy and medium images can be correctly detected, and there are only 3 hard images have one incorrectly detected object in each image.

In the bounding box evaluation part, all of our participants give a 100% on every easy images. Medium images have a 90% average score. For hard images, 14 of them have a 90% average score, 3 of them have a 80% average score, and rest of them have a 70% average score. Note: we use rounding for each image evaluation. If the average score of one image from 10 participants is 84%, we will round it to 80% for the convenience of calculation.

Example of an easy image detection:



Example of a medium image detection:

Example of a hard image detection:

(the original image is too big, so we only take a portion of it)

5.3 ANALYSIS

The goal of our project is to reproduce YOLO v3 algorithm based on Redmon adnd Farhadi's paper[4]. Even though our result is not 100% perfect, it's capable of detecting the objects in our training dataset for most cases. As we dig into those incorrect detections, we found out that the process of non-max suppression is the main reason that leads to errors. Almost all the incorrectly detected images have multiple objects overlapping with each other. As a result, it would be very hard to distinguish one from another in the non-max suppression process. Sometimes it detects two overlapped objects as one and sometimes detects one object as two overlapped objects. We are still working on the solution of this problem.

Besides, we also compare the YOLO v3 algorithm with other algorithms, including SSD513[8], R-FCN[9], and RetinaNet[10]. We directly use those implemented models. Since those models are also well-trained, our comparison focuses on the running time of the detection. We use the same 100 images described in the previous part.

YOLO v3	5.1s
SSD513	14.7s
R-FCN	9.2s
RetinaNet	8.4s

We can see from the result that Yolo v3 is obviously faster than other methods. The difference is subtle for detecting 100 images, but for more images or other real-time application, Yolo v3 undoubtedly would have the best performance.

6 CONCLUSION

Our Yolo v3 object detector achieves a high detection accuracy and has the shortest running time compared with SSD[8], R-FCN[9], and RetinaNet[10]. Generally, we successfully reproduce this fast and accurate detector. Besides, we also did some revisions on the part of thresholding by non-max suppression to get rid of some incorrect detections of our test images. Even though there still exist incorrect detections, the accuracy has improved compared with what we had at first. In the future, we would also work on solutions to the problem we talked about in part 5.3. Besides, we are also looking forward to getting more training data to predict more labels. To do so, we have to play with new datasets and get a new weight for each label, which is a huge amount of work. Specifically, we are trying to combine this detector with a research project which is about the autonomous driving bot. We will use related objects (such as roads, pedestrians, traffic lights, etc) as training dataset. After the detector can recognize those objects, we plan to use it as a helper in our imitation learning model, which can generate actions of the bot based on the current observations.

7 INDIVIDUAL CONTRIBUTION

Zihao Zhu collaborated on researching literature, worked on implementing the Darknet network and prediction layer, testing the model, and analyzing results.

Perry Liu collaborated on researching literature, worked on data processing, implementing pre-trained weights to the model and prediction layer, comparing runtime among different models, researched relevant applications.

We wrote the report together. Our contribution seems equal.

REFERENCES

[1] Amran Hossain, Mohammad Tariqul Islam, Mohammad Shahidul Islam, Muhammad E. H. Chowdhury, Ali F. Almutairi, Qutaiba A. Razouqi, and Norbahiah Misran. A yolov3 deep neural network model to detect brain tumor in portable electromagnetic imaging system. *IEEE Access*, 9:82647–82660, 2021.

[2] Jiwoong Choi, Dayoung Chun, Hyun Kim, and Hyuk-Jae Lee. Gaussian yolov3: An accurate and fast object detector using localization uncertainty for autonomous driving, 2019.

[3] Yi-Qi Huang, Jia-Chun Zheng, Shi-Dan Sun, Cheng-Fu Yang, and Jing Liu. Optimized yolov3 algorithm and its application in traffic flow detections. *Applied Sciences*, 10(9), 2020.

[4] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[5] Joseph Redmon. Yolo: Real-time object detection.

[6] Pjreddie. darknet: Convolutional neural networks. `https://github.com/pjreddie/darknet`, 2022.

[7] Ayoosh Kathuria. Tutorial on implementing yolo v3 from scratch in pytorch, Dec 2019.

[8] uoip. Ssd-variants. `https://github.com/uoip/SSD-variants`, 2018.

[9] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016.

[10] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.