Real-time Object Detection with Tracking Module

Jiahao Cheng, Yi Wai Chow, Zhiyuan Yang

1. Abstract

We rebuild a well-known real-time object detection algorithm, YOLO-v1 (you only look once), and implement a monitor for a specific target using our YOLO-v1 or Redmon's YOLO-v3. The detection model is composed of a convolutional neural network predicting the bounding box and the class. With YOLO's ability to quickly detect objects from a video, we can try to ultimately achieve a real-time efficiency in searching for desired objects and then track it. Our YOLO-v1 model processes images at seven frames per second with a low accuracy of the class. YOLO-v3 performs much better for tracking objects. It could process three frames per second with high accuracy. Compared to Redmon's YOLO, our YOLO-v1 performs well in detecting objects' position, but it makes more classification errors.

2. Introduction & Literature Review

Object detection is one of the significant contents in computer vision and image processing, since it is the fundamental part of intelligent surveillance systems like face recognition, and instance segmentation. Many researchers put effort into making computers processing object detection faster and more accurate. Traditionally, we can detect an object in three stages. Firstly, select some regions by sliding a fixed window on the entire image in different scales and ratios. Then, extract features using SIFT or HOG from these regions [1]. Finally, use the trained classifier such as SVM and AdaBoost for classification [1].

Most popular object detection algorithms based on deep learning can be divided into two categories, namely a one-stage object detection algorithm and a two-stage object detection algorithm. R-CNN is a famous two-stage algorithm that divides the detection problem into two stages. It generates the region proposals containing the object's potential location at first, classifies these boxes, and refines the position of the candidate areas in the second stage [2]. YOLO and SSD (single shot multibox detector) are two representatives of the one-stage algorithms, which use a convolutional neural network to predict the categories and positions of different objects directly [3], but SSD applies multi-scale feature maps with convolutional predictors for detection [4]. In general, the two-stage algorithm has better accuracy, while the one-stage algorithm can quickly detect objects. However, both types of algorithms have been improved to balance the trade-off between speed and accuracy. With the development of research and technology, both algorithms can achieve better performance in speed and accuracy.

Although other object detection algorithms have high-quality performance in both speed and accuracy and the latest optimized YOLO, YOLO-v4 [5], has been published, we decide to rebuild YOLO-v1 and use our model and Redmon's YOLO-v3 to detect and track objects. First, the structure of the YOLO-v1 model is simpler. CNN takes the resized images as the input and outputs bounding boxes and class probabilities directly, which is convenient for training and testing. In addition, the one-stage algorithm is relatively faster so that we can give more time for tracking and matching objects since matching two targets is time-consuming.

Our goal is to detect objects from images and videos in real-time using our YOLO-v1 model with good accuracy and using it for tracking objects quickly in a video with high accuracy. More details about tracking and object searching will be introduced in later sections on this paper.

3. Methodology

We construct the YOLO-v1 model based on Redmon's YOLO-v1 [3]. The input image would be resized to 448 × 448, and put into a 4-dimensional tensor with the shape (1, 3, 448, 448) as the input of the CNN. The neural network is generated according to the predefined configuration file including hyperparameters of the neural network as well as the parameters of each layer. We get this idea from Chaurasia's description [6]. In our model, we use Redmon's yolov1.cfg file [7], constructing a CNN which has 24 convolutional layers with four max pool layers in between, followed by one fully local connected layer and a dropout layer with the other fully connected layer. Since we need the output values from 0 to 1, one sigmoid layer is connected at the end [3]. In addition, for the layer that requires the linear activation function, we use the leaky rectified function with a negative slope of 0.1. The output is an $(S, S, B \times 5 + C)$ tensor, where $S \times S$ is the number of regions among the input image for detection, B is the number of bounding boxes for prediction, and C represents the number of classes. Since we use PASCAL VOC 2007 image dataset [7] for training and testing, $C = 20$. We set $S = 7$ and $B = 3$, resulting in the size of $(7, 7, 3 \times 5 + 20)$ for the output.

3.1. YOLO-v1 Detection

After we get the tensor output, our system analyzes the result and draws the bounding box with its label and probability for each predicted object. It is designed based on Redmon's YOLO source code [8] and Kimura's YOLO-v1 code [9]. First, it decodes the tensor result to get the correct position of the bounding boxes and the class that has the highest probabilities among 20 categories which are higher than the probability threshold, since the original result stores x, y, relative to the bounds of the grid cells from 0 to 1, and w, h, relative to the whole image between 0 and 1.

Then, the system filters predicted boxes based on the confidence threshold and non-maximal suppression threshold to delete predictions in which the intersection over union value between bounding boxes are below the non-maximal suppression threshold and predictions with a low confidence value. It is a useful and significant process that increases accuracy. We use the built-in non-maximal suppression function from the OpenCV library since it is well-designed and takes vector inputs that can speed up the computation. Thus, the system gets the final result and uses these values directly to plot boxes to the image.

3.2. Training

We use the sum of squared error as our loss function based on Redmon's design [4] in which it increases the weight of the bounding box values (x, y, w, h) and decreases the weight of the confidence that does not contain objects. In addition, in order to balance the impact of the large objects on small objects, we use the square root of the width and height for the loss function. Thus, the loss function is:

$$
\begin{aligned}
\mathrm{Loss}(\mathrm{pred},\, \hat{\mathrm{real}}) = {} & \lambda_1 \sum_{i=0}^{S^2} \sum_{j=0}^{B} \prod_{i,j}^{obj} [(x_i - \hat{x_i})^2 + (y_i - \hat{y_i})^2] \\
& + \lambda_1 \sum_{i=0}^{S^2} \sum_{j=0}^{B} \prod_{i,j}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w_i}})^2 + (\sqrt{h_i} - \sqrt{\hat{y_i}})^2] \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \prod_{i,j}^{obj} (C_i - \hat{C_i})^2 + \lambda_2 \sum_{i=0}^{S^2} \sum_{j=0}^{B} \prod_{i,j}^{noobj} (C_i - \hat{C_i})^2 \\
& + \sum_{i=0}^{S^2} \prod_{i}^{obj} \sum_{c=0}^{C} (p_i(c) - \hat{p_i}(c))^2
\end{aligned}
$$

where λ_1 is the weight of the bounding box values and λ_2 is the weight of the non-object confidence [3]. According to the yolov1.cfg file [9], we set $\lambda_1 = 5$ and $\lambda_2 = 0.5$.

We do not pre-train our convolutional layers since Redmon's paper mentions that they took approximately one week for that [3]. It is late when we realize this process is significant and necessary to get precise results. Instead, we train our model using the PASCAL VOC 2007 image dataset [7]. For the dataset we used, we split the PASCAL VOC 2007 image dataset into a 7/3 training to validation dataset. In the training process, we train a total of 75 parameters using SGD as the optimizer. Following the paper training procedure, we train the network for about 135 epochs with momentum of 0.9 and a weight decay of 0.0005. Due to hardware constraints, our batch size is limited to 10 instead of the 64 suggested by the paper. In terms of the learning rate, we update it based on the number of epochs the net went through. At epoch 0, we set it to be 3e-04 for 75 epochs. Then continue to be 1e-04 for another 30 epochs. Lastly, it is set to be 1e-05 until the end of the training process.

In each of the training batches, the Yolo net predicts a result vector and this result vector is again passed into the Loss function module, with the loss function, we can calculate the gradient and the SGD optimizer can update weight using the gradient. After the Yolo net completes all the batches training, we can evaluate the error rate of the trained model in this epoch and decide if we keep this as our final trained model.

3.3. Tracking

Our ultimate goal for the tracking module is to be able to search for the desired object provided by the user and then track it in real-time. This spotting and tracking pipeline can be divided by three milestones: 1) being able to detect the positions of objects, 2) being able to track the positions of objects and 3) being able to track objects that match the image provided by users. The first stage is done using the Yolo net, so we will spend more time discussing the subsequent stages.

The tracking algorithm takes the advantage that the Yolo net outputs the centroids of objects together with a bounding box that describes where an object appears. This means we have areas of interest being spotted already, and the computation for centroids of objects can be saved. Since we have the centroids of each object, we can use the one simple but very effective heuristic: the coordinates of the same object in two consecutive frames of a video will not be far away from each other[11]. From this heuristic, we can construct a very computationally efficient algorithm: compute the euclidean distance between each pair of object centroids and label the closest pairs as the same objects.

This algorithm is extremely fast, but it has several drawbacks. First, if an object hides itself (for example, a car passes a person from his front) for even a single frame, the tracker will lose track of it and it will be labelled as a new object at the time it re-appears in the video. This problem can be fixed by setting a disappearance counter for each existing object which keeps the label and previous location of an object when it disappears on the one frame. If an object disappears for enough number of consecutive frames, then its information will be deleted from the tracker. Otherwise, its information will be picked up again when it comes back. This solves the re-labelling problem, but it introduces another problem. If an object disappears and a different object enters the screen, then the latter object will pick up the information that belongs to the previously disappeared object. This problem cannot be completely solved without the usage of image matching algorithms, but it can be alleviated using simple movement prediction algorithms. That is, we can use the locations of a tracked object in two consecutive frames to compute a velocity vector and add this vector to the current location of the object to predict where it should be in the next frame. Then, compare the predicted location with the location of the newly appeared object to see if they are close enough to each other.

After we achieved the functionality of tracking objects, we will move on to our next stage: recognize the desired object. Recognition of desired objects is not hard to do, especially we have the Yolo net labelled the areas of interest in each frame. However, it is hard to do in real-time. The first idea we tried is to use SIFT to find the key-points of each object detected in every frame, and match the key-points with the ones on the desired image of the object. This process is very robust. That is, we can achieve a very high rate in successfully recognizing the right object. However, such a process is very computationally expensive. The average time for processing one frame using such a method is 0.8 seconds, which has a severe impact on the FPS. Therefore, we use two other methods for recognizing the desired image: 1) correlating the areas of interest with the desired image on each image or 2) compute a structural similarity score between the areas of interest and the desired image. The second method is faster than the first method, but its rate of successfully matching objects is not as good as the first method. So, we decided to implement both methods and let the user pick the preferred one in different situations.

These two algorithms are faster than using SIFT, but they still have a poor FPS in practical use. Thus, we augmented the image matching algorithm by exploiting the fact that the Yolo net can give us the label of each object. That is, we can feed the desired image into the Yolo net to get a label of the wanted object. After we get the label, we can check for the presence of the same labels in one frame. If there is no object having the same label as the desired object, then no computation on correlation or structural similarity needs to be done. Implementing such a simple idea has a great improvement in the efficiency of the tracker, and the average FPS has almost been doubled.

4. Experiments

4.1.1. Detection

Since the overall performance in the accuracy achieved is suboptimal, we tried several loss functions by tuning the lambda values, squaring or taking the square root of the coordinate values and probabilities. By giving more weights for the loss of class probabilities, the results did not improve. It mostly outputs the incorrect classification or the bounding box is extremely large or small.

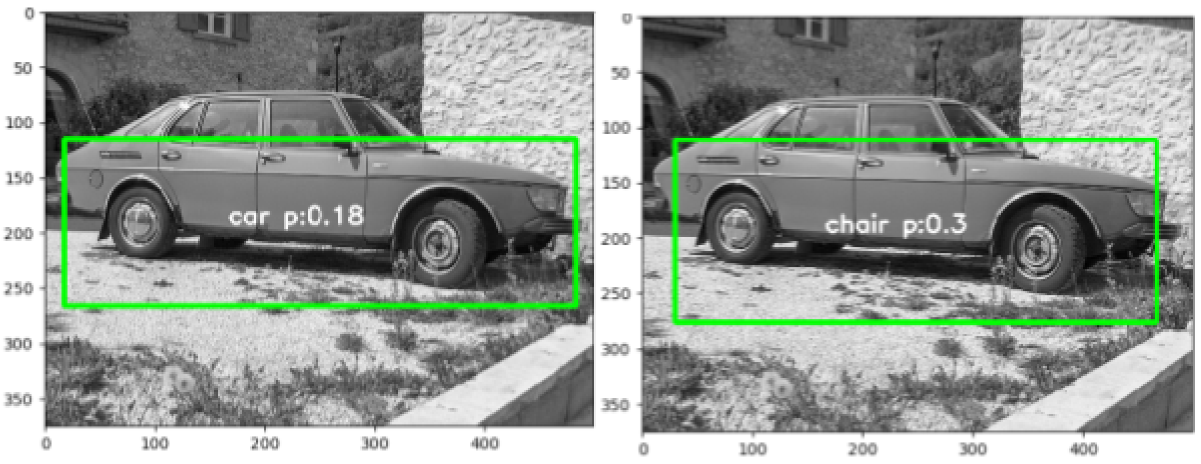

Figure 1: Car Results. The left result using the loss function we defined before. The right result using the loss function which gives equally weights to class probabilities and coordinates.

4.1.2. Adam Optimizer

We tried the use of Adam Optimizer for training, but due to hardware constraints [*1] and the size of the Yolo net architecture [*2]. we are unable to effectively utilize the adam optimizer.

[*1]: we are using a single RTX 2070 GPU for computation purpose
[*2]: The Yolo net consists of 24 convolution layers, 4 max pool layers, two linear fully connected layers and 1 dropout layer. All of these combined require a lot more computation power than what we have when using Adam Optimizer.

4.1.3. Learning rate

In the YOLO's paper, it is suggested to use a high learning rate, such as 0.01 to initialize the training process. However, when applied that learning rate to our version of the Yolo net, the high learning rate would cause instability for the weight update process and thus outputting nan values.

4.2.1. Tracking

Since the tracking module uses the information of objects obtained from the detector, its accuracy is highly related to the accuracy of the detector. Therefore, to make the experiments less biased and show only the functionality of the tracking module, we decide to do the experiments by mounting the tracking module on two correctly implemented package trackers: Yolov3 and Yolov3-tiny. Due to limitations on the hardware, we have the following experiments performed on CPU.

Experiment 1:

The first experiment focuses on investigating the efficiency of the tracking module. Below is a table showing the average FPS of the different detectors with or without the tracking module mounted running on one sample testing video:

Tracking?	Yolov3 Average FPS	Yolov3-tiny Average FPS
Without tracking	5.71 FPS	53.31 FPS
With tracking	4.53 FPS	31.52 FPS

From the above results, we can see that the tracking module has introduced a constant computation time for each frame. Such a constant computation time has a dominant effect on the average FPS when using Yolov3-tiny as a detector because the computation time is relatively large compared to the processing time of the detector. In contrast, the Yolov3 net is spending a lot of time processing each frame, so the constant computation time introduced by the tracker has less effect on the average FPS. With this additional computation time, we can still see a relatively fluent video with the tracking module mounted on fast detectors.

Experiment 2:

The second experiment focuses on investigating the efficiency of the object searching functionality used in tracking the desired object. We would like to see how efficient it is:

Object Searching?	Yolov3 Average FPS	Yolov3-tiny Average FPS
Tracking with searching	1.55 FPS	2.36 FPS
Tracking without searching	4.53 FPS	31.52 FPS

The result is pretty bad here. Such a huge dropping in the average FPS indicates the usage of a bad method for searching objects. The object searching method used here by correlating the areas of interest with the desired object. This introduces the necessity of using a faster searching algorithm, so we have performed our last experiment.

Experiment 3:

The third experiment we performed focuses on investigating the efficiency of different searching algorithms. We have tried three metrics in searching the objects: correlation (benchmark), structural similarity, and the sum of squared distance. The third metric has a similar computation time as computing the structural similarity score, but it is very sensitive to noise and thus has poor accuracy. We do not include that metric in the following table.

Metrics:	Yolov3 Average FPS	Yolov3-tiny Average FPS
Correlation	1.55 FPS	2.36 FPS
SSIM	3.11 FPS	20.07 FPS

This suggests that the structural similarity score is an efficient object searching metric to use.

5. Results

After these adjustments, we get the final YOLO-v1 model. For testing, we set the confidence threshold equals 0.1, the probability threshold equals 0.1, and the non-maximal suppression threshold equals 0.35. In general, our model has an approximately 43% chance of failing to detect the target by testing 5,011 VOC 2007 images. For pictures where the target is found, the model has relatively 70 % accuracy to get proper bounding boxes, while the accuracy of the classification is around 20%. We get the accuracy results by randomly selecting 20 independent and identically samples. Figure 2 and 3 show some results. Although it gets the correct bounding boxes and classes, the model underperforms for finding multiple objects when they are clustered.

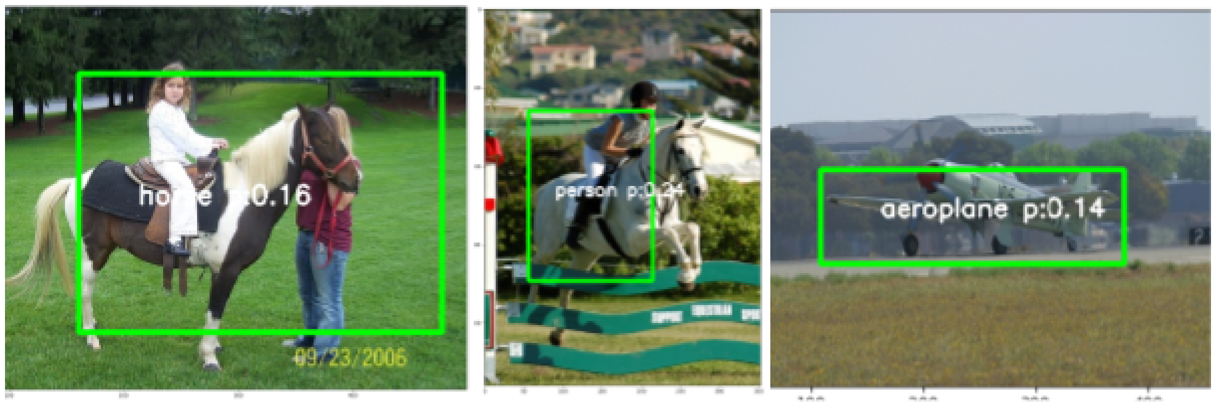

Figure 2: Good Results. Outputs with the correct class and good bounding boxes using our YOLO-v1 model.

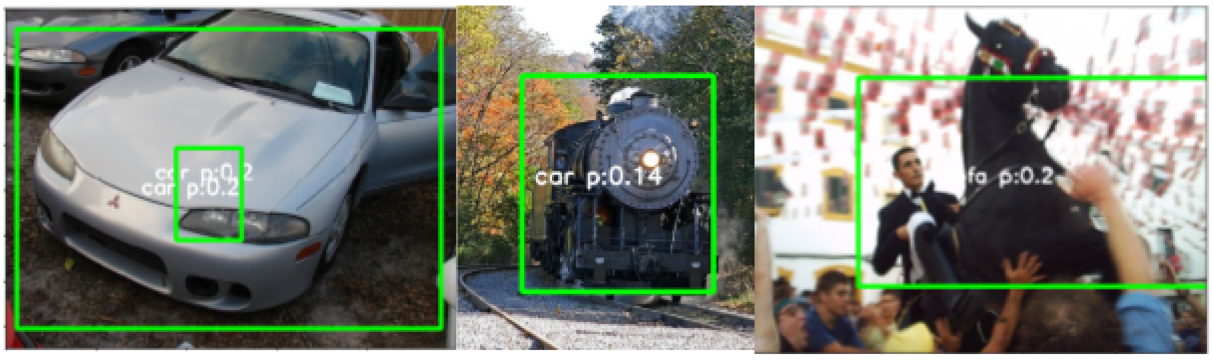

Figure 3: Poor Results. Results that has poor box coordinates or labels.

The tracking module has achieved relatively real-time efficiency. Its rate of successfully tracking a desired object is almost 100%, which means if the detector can find such an object in a video then the track will be able to spot it and track it. The following figure shows examples of successful target locating using Yolov3 with a SSIM threshold of 0.7 and 0.55 respectively:

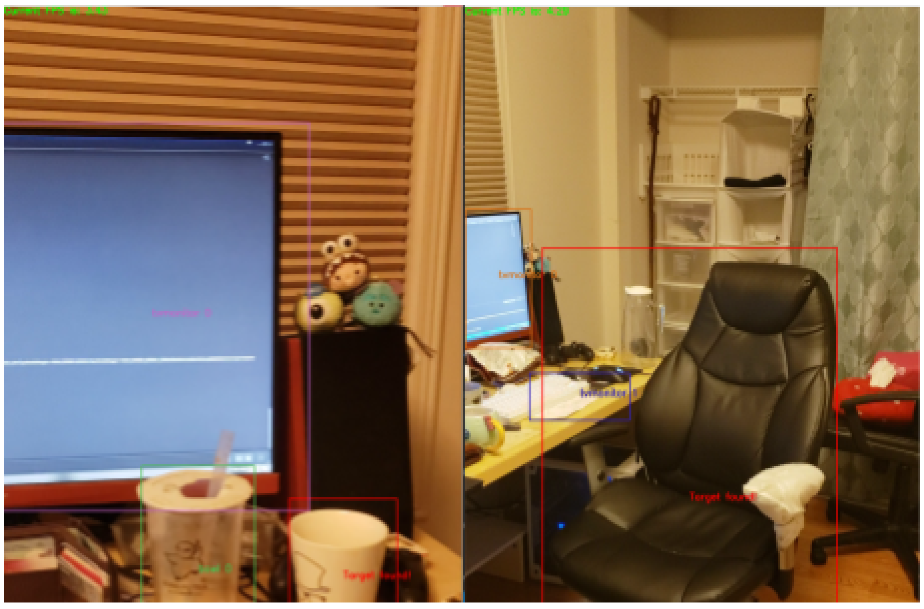

Figure 4: Targets have been successfully found.

6. Conclusion

We introduce our YOLO-v1 model and the method of tracking a specific object for the video. The model is designed based on Redmon's YOLO-v1 mostly, except the pretraining process. The tracking algorithm tracks and matches objects by comparing correlations in different grids.

Our YOLO-v1 underperforms in class accuracy. While it can detect most objects' position in images, it struggles to precisely identify the class of the objects. One of the reasons could be we did not use a pre-trained model as our net's backbone. Since a well-performed CNN requires a large amount of dataset, training thousands of images is not enough to generalize the model.

The tracking module is shown to have a good efficiency and accuracy with the use of good detectors, achieving a relatively real-time object searching and tracking functionality.

7. Authors' Contributions

All of us participated in the design and implementation of the YOLO-v1 model. Jiahao Cheng implemented the YOLO network (yolo.py & util.py) and detection part (detection.py). Yi Wai Chow designed the data encoding from the VOC 2007 dataset (DataSet.py) and led the training process (training.py). Zhiyuan Yang implemented the loss function (loss.py). Everyone contributed to the design and correction of the tracking algorithm. Zhiyuan Yang directed and mainly worked out tracking implementation (tracker.py), and performed detection using the YOLO-v3 model (detection.py & yolo_cv.py). Yi Wai Chow conducted the experiments and tuned the hyperparameters of the models. Jiahao Cheng aided in interpreting the results. All authors discussed the results and worked on the report.

8. References

[1] T. Malisiewicz, A. Gupta and A. A. Efros, "Ensemble of exemplar-SVMs for object detection and beyond," *2011 International Conference on Computer Vision*, Barcelona, pp. 89-96, doi: 10.1109/ICCV.2011.6126229. 2011.

[2] P. F. Felzenszwalb, R. B. Girshick, D. McAllester and D. Ramanan, "Object Detection with Discriminatively Trained Part-Based Models," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627-1645, doi: 10.1109/TPAMI.2009.167. Sept. 2010.

[3] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, pp. 779-788, doi: 10.1109/CVPR.2016.91. 2016.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg. "SSD: Single Shot MultiBox Detector." Lecture Notes in Computer Science: 21–37. *Crossref.* Web. 2016.

[5] A. Bochkovskiy, C. Wang, and H.M. Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." *ArXiv*, abs/2004.10934. 2020.

[6] A. Chaurasia. Yolo-V3. https://github.com/AyushExel/Detectx-Yolo-V3. Accessed: 2020-7-23.

[7] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. "The pascal visual object classes challenge: A retrospective." *International Journal of Computer Vision*, 111(1):98-136, Jan. 2015.

[8] J. Redmon. Darknet: Open-source neural networks in C. http://pjreddie.com/darknet/, 2013-2016.

[9] M. Kimura. YOLO-v1 Pytorch. https://github.com/motokimura/yolo_v1_pytorch. Accessed: 2020-7-25.

[10] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600-612, Apr. 2004.

[11] A. Rosebrock. "Simple object tracking with OpenCV". *Py Image Search.* https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/. July 23. 2018.