You Only Look Once (YOLO) Optimization and Analysis

Implementation Track
4 Group Members

Abstract

Objective detection has been a major part of computer vision and it is widely applied to fields like self driving cars, face recognition and image modifications. Multiple algorithms have been invented to improve the performance of object detection. Our group implemented the You Look Only Once (YOLO) [1] Object detection algorithm introduced by Joseph Redmon et al. We reimplemented the algorithm using the PyTorch[12] library and then based on the original version, we tried different configurations to see if the result could be optimized. In this report, we will discuss why YOLO may be useful and what other contemporary works have done in comparison. Then we will explain the mechanism behind YOLO and the results we get from the original YOLO implementation. We also attempted to optimize YOLO with other backbones in hopes of making it more viable.

1 Motivation

Object detection has a wide range of usage in computer vision tasks, with use cases ranging from automated driving cars to object tracking in sports matches. It allows us to identify objects and their locations in images or videos. Here, we provide two examples of real-life applications of object detection. The first usage is in CCTV surveillance. This system plays a important role in security of our daily lives. Usage of object detection in CCTV surveillance system can help reduce the memory usage. For example, starting screening when certain objects are detected. The second usage lies in vehicle detection. One way we can use object detection is to track a vehicle's license plate and associated accidents related to that vehicle. Another more widely known application is called advanced driver assistance systems (ADAS), which allows vehicles to detect driving lanes and pedestrians. These, among others, show us that object detection plays a significant role in our daily lives.

2 Problem Statement

One may confuse object detection with image recognition. However, these two are distinct tasks since image recognition does not provide bounding boxes for each image and only predicts one class for each image. Consequently, object detection is more complex in nature.

There are usually two approaches to object detection: Hand-designed feature extractor and deep learning. The deep learning approach uses Convolutional Neural Networks for training and does not involve additional steps to define features like the former approach does; thus, in recent years it has become widely adopted.

The basic structure of an object's detection in a deep learning problem can be described into two parts. The first part is an encoder, which takes a set of images as input and outputs extracted statistical features. The second part is a decoder used to classify objects and predict bounding boxes. One example of a simple decoder could be a regressor such as an SVM. However, a regressor requires specifying the numbers of objects in an image, meaning if the model is designed to predict one class

then it cannot output two classes given a two-class image. Therefore, later researchers replaced the regressor with a regional proposed network, which takes the input from the encoder and then outputs a set of potential regions feeding to a classification network to output image classes [2]. In recent developments, researchers combine these procedures into a single set of neural networks in order to reduce time complexity. YOLO is one of such developments [1].

3 Related Work

RCNN An effective object detection neural network developed before YOLO is RCNN [2], which combines regional proposal with convolutional neural networks. If we want to use normal CNN to do object detection, the number of regions where we want to perform classification is too large to be usefully computed. To improve the efficiency and quality of region classification, RCNN promotes selective search on regions in an image. The RCNN then wraps all the region proposals and feeds them into a CNN, which extracts features from the regions. The features are classified by SVMs. Besides the classifications, the CNN also predicts offset values to optimize the bounding boxes for the proposed regions. A huge problem with RCNN is its computational time. Due to the fixed selective search algorithm, RCNN still needs a large number of proposed regions to avoid the effects of some outliers. Running CNN on each proposed region would be prohibitively expensive.

Fast RCNN To solve this problem, Girshick et al. suggest Fast RCNN [3]. Instead of running CNN on each region, only one CNN would be run on each image. The CNN will generate a feature map of the image and the proposal regions will then be generated based on the feature map. The time required for running the CNN is significantly reduced but the time for running region proposals still impacts the performance significantly. The selective search algorithm does not have learning ability, so the time could not be further optimized during execution.

Faster RCNN Shaoqing et al. addresses the problem of slow region proposal algorithm by introducing a new algorithm called Faster RCNN [4]. It uses a network to predict region proposals so that the slow and fixed selective search algorithm can be eliminated. In addition, the network can learn and improve its accuracy during execution. The Faster RCNN improves the performance by up to ten times. Still, since all the RCNN algorithms propose regions, they miss the ability to look at the whole image. Additionally, RCNNs often consider background patches as objects [1].

YOLO The YOLO algorithm predicts both the bounding boxes and the object classification at the same time from the same network. Though it may have lower accuracy than the RCNN family, it is less vulnerable to background errors. Further, it reduces the computation time in a scalar manner. As the researchers claimed, the new architecture is able to process images at up 45 frames per second [1] on their hardware.

4 Methodologies

4.1 Overview of YOLO Pipeline

YOLO tries to detect C different classes of object by dividing the image into S cells uniformly and produce B bounding boxes that each indicate object location within a cell. At training time, a batch of images with its ground truth bounding boxes is fed into the neural network. The network will output a tensor of size S,S,(C+5*B), which carries its predictions. The prediction tensor is used to calculate loss and propagates updates to filter weights in the backward pass.

At test time, an image without ground truth bounding boxes is fed into the network, and the network will produce a tensor with the same dimensions used in training that carries its prediction of bounding box locations, classes, and confidences.

4.2 Network Architectures

In the YOLO V1 [1] Paper, the model's network architecture can be treated as two modules: the detection network which consists of convolutional layers in the front to capture features of the image and several fully connected layers that utilizes flattened output of the last convolution layer to produce

prediction. In [1], the author used Darknet [11], an open source neural network library written in C to implement their original detection network. Since our implementation will use PyTorch [12], for the sake of better weight initialization and lower time investment, we decided to add VGG16 [12] and ResNet152 [12] as options for the detection network (Pytorch has pretrained weights of VGG and ResNet on ImageNet [13] dataset).

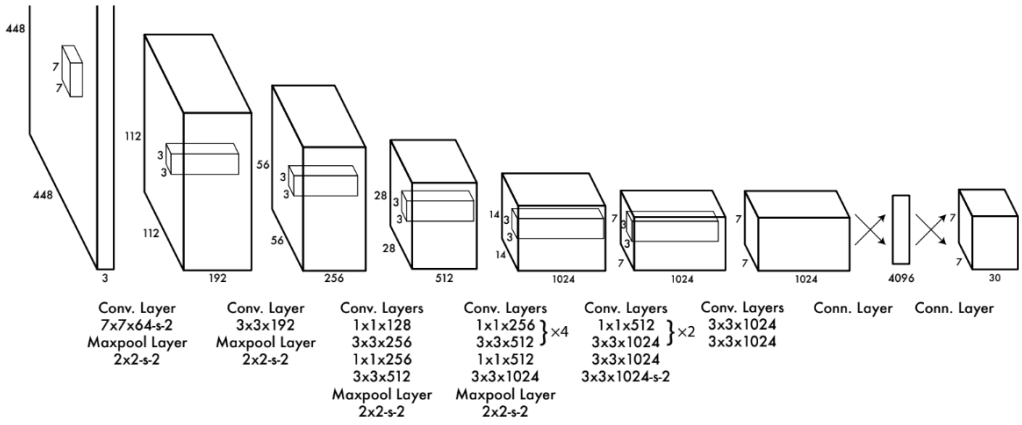

Figure 1: YOLO CNN Architectural Overview

Original Darknet The detailed architecture setup of this variant of detection network is shown in Figure 1. It consists of interleaved convolutional layers and max pooling layers. Using the example of the first layer, 7x7 means the receptive field has size 7x7x3 in width, height, and channel. 64 means we will have 64 filters (consider each filter as a set of weights). Generally, we will have more filters towards the end of the detection network compared to the start. 's-2' means we will have stride of 2. So instead of sliding the receptive field by one pixel, we do it in 2 instead. To help convergence, for each convolutional layer, we also append batch normalization layers which are not mentioned in the original paper before LeakyReLu activation functions.

Our Modified Version 1: Group Norm Darknet This is a variant of the original detection network we mentioned in the above section. Instead of using batch normalization, we used group normalization [8], setting the number of groups to the input batch size divided by 2, so we had (batch size/2) groups. For group normalization, standardization (subtract mean and divided by standard deviation) is performed within each group instead of the whole batch. Group normalization can provide faster convergence and even higher accuracy in prediction stage [8].

Our Modified Version 2: ResNet152 ResNet [9] is a well-known image classification network that has good performance on the ImageNet [13] dataset which consists of 1000 classes. Since our focus will be implementing YOLO, we directly used the pretrained weights of ResNet152 that were provided natively by PyTorch.

Our Modified Version 3: VGG16 VGG [10] is another prominent image classification network that has decent performance on ImageNet. We directly use pretrained VGG16 weights with batch normalization layers provided by PyTorch.

Bounding Box Prediction Layers The above mentioned architectures can all perform the feature detection task. Their output from the last layer is flattened into a one dimensional tensor and fed into a fully connected layer to make the final prediction. In the original paper [1], the flattened layer has size S*S*1024. We chain this with a linear layer with S*S*1024 inputs and 4096 outputs, a dropout layer of 0.5, a LeakyReLu of 0.1 and a linear layer with 4096 inputs and S*S*(C+5*B) outputs. Notice this has the same dimension of the prediction tensor we mentioned earlier in Section 4.1. For ResNet152, we replace the fully connected layers with a single linear layer that has 2048 inputs and S*S*(C+5*B) outputs. Because ResNet's classification layer already has dropouts, we did not bother

adding dropout again. With similar reasoning for VGG16, we replaced the fully connected layers with a single linear layer with 4096 inputs and S*S*(C+5*B) outputs.

4.3 Loss

The loss function of YOLO is quite complicated. Its full form is shown in Figure 2. To break this loss function down, we will explain the intuition behind different components. First, let us break down the structure of the network's output: the tensor of shape [S,S,C+5*B]. As a brief recall, S is the number of cells we are dividing the image into, C is the number of classes for the objects, and B is the number of bounding boxes we are allowed to have per cell. Starting from index 0, index 2 ranges from 0 to C+5*B, and the first C elements indicates class score which is the per class probability p(c), the following elements in a group of 5 is the bounding box information arranged in [confidence C, x, y, width, height]. Here x represents the horizontal axis of the bounding box's center, y represents the vertical axis of the bounding box's center. This coordinate information is all normalized against the width and height of a cell so the value will be within 0 to 1. However, width and height can be greater than 1 since there can be bounding box that spans across multiple cells. The Loss of YOLO is a combination of several sum-square errors.

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

Figure 2: Mean Squared Error Objective Function[1]

Identify Function 1_i^{obj} indicates whether there is object present in cell i in the ground truth, and 1_{ij}^{obj} evaluate to 1 if bounding box j has highest confidence, highest intersection over union, out of all bounding box predictions in that cell.

Bounding Box Error The first two line of Figure 2 describes the Bounding Box Error. x_i and y_i are the x and y axis of the i^{th} bounding box's center. The first line is calculating the sum-square error of the center of the bounding box with highest IOU among all other bounding boxes in that cell. The identify function will make sure only cells with objects are counted and only the bounding boxes with the highest IOU will be counted. The w_i, h_i are the width and height of the bounding box. The logic is the same as that for the x and y coordinates but the width and height are square rooted. Since the sizes of different bounding boxes might vary greatly, we must square root the values as even small differences in predicting large bounding boxes would exceed the weights of the difference in predicting small bounding boxes. The square root will balance the importance of bounding boxes with different sizes. Also, since compared to the classification error, we want to be more precise on localization error, we apply $\lambda_{coord} = 5$ to the start of the first two terms.

Confidence Error There are two components in the Confidence Error parts, the third and forth line in Figure 2. The third line calculates the loss of confidence in bounding boxes of each cell that has objects. This is similar to the bounding box error except only the bounding box with the highest

IOU will be counted. However, it is not enough to only consider cells with objects. We also need the fourth line to take cells without objects into consideration for a better performance but since there are normally more cells without objects and we do not want the loss of them to overpower the loss of cells with objects. Thus we use the constant $\lambda_{noobj} = 0.5$ to lower its priority.

Class Error The last line of the loss function is the pure classification term that calculates the errors for all classes. We only consider the accuracy of cells that have objects in them. And we sum the errors of predictions for each class.

5 Evaluation

This implementation track project centered around the replication of the YOLO algorithm. In the evaluation of our model, multiple different setups were used to parallelize training given the limited time. The primary test setups utilized were Google Colab using a P100 GPU, local CUDA-Capable machines, and The University of Michigan Greatlakes computing cluster (less frequently due to load issues). Several learning rates were tried and the ones presented consistently gave the best results. To verify its correctness we tested our implementation against the PASCAL VOC 2007 and 2012 datasets used in the original Redmon paper [1].

5.1 Metrics

We used the mean average precision (mAP) to measure the performance of our models. It is the mean of average precision across all object classes. AP can be seen as the area under curve region size of the precision vs recall curve. In the context of object detection, a bounding box with correct class, and intersection over union (IOU) with ground truth higher than 0.5 is considered as True Positive. Intersection over union is defined as the intersection region of two bounding boxes divided by their union.

5.2 Data Processing

Each dataset required preprocessing of some manner before it could be fed into our model. Each dataset came annotated with additional information which was scraped and compiled into a useful formulation for YOLO. Primarily, this entailed translating subsets of the annotation information into bounding boxes and classifications with a subsequent processing of all images to assign classes to each bounding box appropriately. Additionally, multiple types of data augmentation were tested including scaling, horizontal flipping, blurring, color hue, saturation adjustments, and translation. For the PASCAL VOC dataset, we used the given training and validation sets. We investigated two novel datasets: MakeML Cars and Roadsigns and Roboflow's Mask Wearing dataset. For these two datasets, a 90:10 training:validation split was randomly generated. Due to time considerations, pre-trained weights were used for initialization over random or zero initializations for each dataset.

5.3 PASCAL VOC 2007 & 2012

This dataset was used in the original paper and was used for correctness testing during this project. PASCAL VOC [5] features numerous classes ranging from bottles to planes and thus serves as a general standardized dataset for object class recognition. The original YOLO implementation scored well but not best in class with approximate accuracies of 65.5% mAP on the 2007 set and 57.9% mAP on the 2012 set [1]. Due to the lack of computational resources, we trained on the 2007 and 2012 datasets then validated exclusively on 2012. We swept our hyperparameters on a provided 100 sample train and validation split in an attempt to find a setup that would yield useful results quickly.

Figure 3: PASCAL VOC Train Predicted Bounding Box and Class on Test Image

The results are based on a 20 class subset of PASCAL VOC. Multiple backbones were used to validate our implementation. The best achieved mAP was 49.36% on the ResNet152 framework indicating that our implementation was approximately 8% worse than the original YOLO implementation on the same validation set as the original paper. The VGG16 framework also seemed to validate a correct implementation but performed 11% worse than the ResNet152 framework on average. You can see the results of our ResNet152 framework implementation in Figure 4 as well as a bounding box prediction for a Train on the test data in Figure 3.

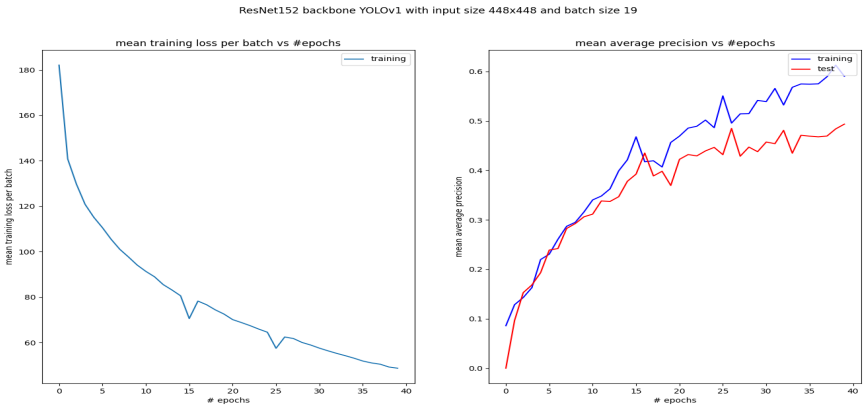

Figure 4: Left shows the mean loss versus epochs. Right shows the train and test mAP versus epochs.

Backbone	Dataset	Batch/Image Size	Learning Rate	Test mAP	Train mAP	Last Loss over Batch
ResNet152	VOC	19/448	1e-4 x 15, 1e-5 x 10, 5e-6 x 15	0.4936	0.6132	48.68
VGG16	VOC	19/448	The same as above	0.3866	0.5137	61.24
Original	VOC	19/448	The same as above	0.0842	0.1620	94.20
ResNet152	Mask	12/224	1e-5 x 300, 2e-5 x 300, 5e-5 x 234	0.0113	0.8642	34.04
VGG16	Traffic	19/448	1e-5 x 10, 2e-5 x 15, 5e-5 x 5	.0063	0.0732	104.09

Table 1: Performance Statistics and Hyper Parameters

The hyperparameters used for all tests can be found in Table 1. Of note, the original architecture using the DarkNet backbone performed relatively poorly. One of the major reasons is that, in the original paper, the detection network was trained in ImageNet for a week before it is fine-tuned on PASCAL VOC for the object detection task. In our implementation, we started the training from scratch instead. Without a systematic strategy to find the optimal learning rate setup, it is expected that it can't converge within 40 epochs. In addition, the original paper used batch size of 64, however, we can only handle batch sizes of 19 due to the limitations of the hardware available to us. Batch size can influence convergence speed substantially which can be seen in our novel dataset analysis.

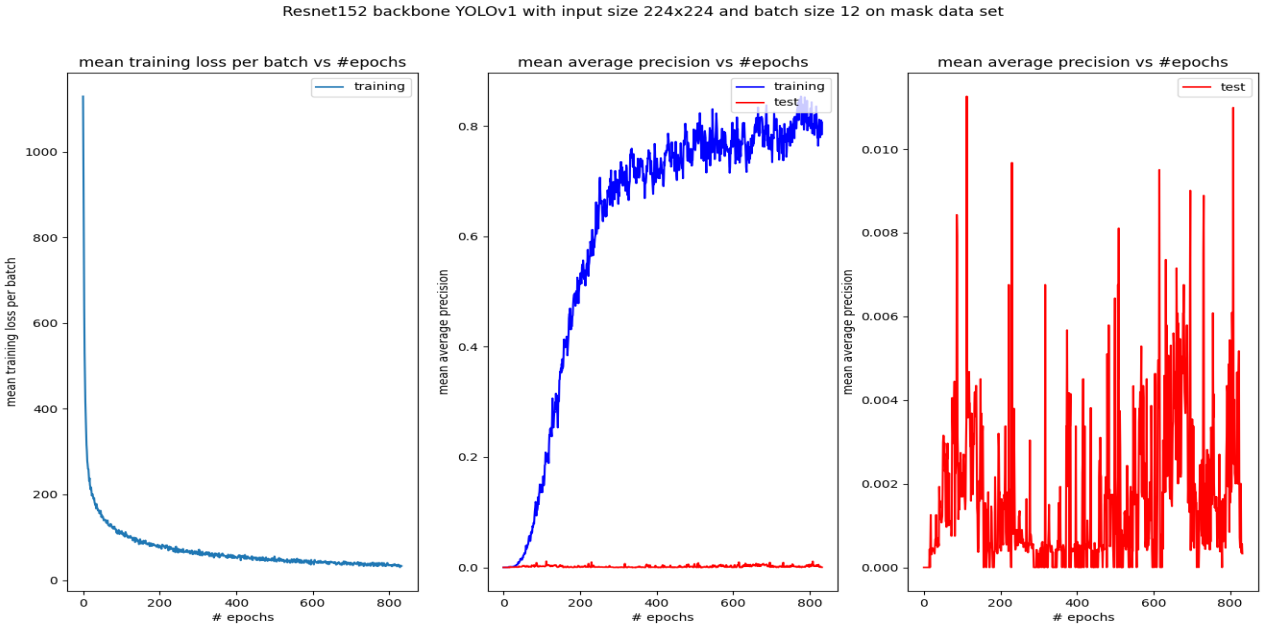

Figure 5: Left shows the loss versus epochs. Center shows the train and test mAP versus epochs. Right shows the volatility of mAP versus epochs.

5.4 Novel Datasets

Two additional novel datasets were explored for the purposes of evaluating our YOLO implementation. The first, Roboflow Mask Wearing dataset [6], is a brand new small 2020 compilation of people's faces wearing masks with corresponding class annotations. The second dataset, MakeML Cars and Traffic Signs [7], has 10,000 annotated images featuring 10 classes relating to traffic including pedestrians, signs, cars, and trucks. The performance using each framework on these novel datasets was quite poor. The best results were achieved on the mask wearing dataset and the MakeML dataset using the ResNet152 backbone. These results can be seen in Table 1 and as well as Figure 5. As image augmentation did not provide substantive improvements in test mAP, these results are without augmentation but group norm did provide a significantly faster convergence.

Though the train mAP was very high, indicating our implementation was sound, the test mAP was volatile and extremely low. There are multiple reasons we believe the performance is significantly degraded compared to PASCAL VOC including a naive hyperparameter sweep, poor weight initialization, significantly smaller images, and clusters of small objects grouped together that, as noted in [1], YOLO performs poorly with. We hoped that significant image augmentation would improve YOLO performance and bridge the divide to more modernsolutions but it did not. Additionally, YOLO only allows 1 object per cell, 7x7 cells are not enough for over 40 bounding boxes in a small patch of an image. Further, we believe that each dataset will require its own tuned set of hyperparameters that we were unable to deduce without a more stringent and methodical approach. We verified the implementations were sound as the train mAP was high regardless of dataset as can be seen in Figure

5 for the Mask dataset. The traffic dataset, similarly, had extremely high train mAP but low test mAP. Unfortunately, the latest runs of the Traffic dataset crashed so the results are old 20 epoch results rather than new several hundred iteration results.

6 Conclusion

The YOLO has a specific advantage in object detection. Compared to other methods that separate the CNN and the region selection steps, YOLO trains the localization features and the classification features at the same time and in the same neural networks so YOLO does not bias on any particular regions in an image. Also, since only one CNN is involved, YOLO has a relatively good running speed compared to other object detecting algorithms. As an example, a GTX 1060m was able to run our YOLO implementation at 56 FPS. Unfortunately, YOLO suffers a variety of drawbacks that we were unable to rectify. For example, if its cell number is not large enough, it cannot focus on details as much as methods like RCNN. Consequently, YOLO is weak in detecting very small objects that are tightly clustered. Datasets with mostly small objects that are tightly grouped should not rely on YOLO as a first option.

Another problem with YOLO is also a common problem of all object detection algorithm. YOLO is strict with data preprocessing. The training data needs the location of bounding boxes and also the classifications to be specified. Moreover, the bounding boxes could not be too small since YOLO is not well designed for small object detection. It is also a major challenge that we are facing while training YOLO using different datasets. Additionally, YOLO seems to be highly sensitive to the hyperparameters and, without a systematic way of selecting these parameters, struggles with adapting to a new dataset without careful tuning. We do not get expected results in either of the novel datasets we tried though we do using the same dataset in the original paper. The Traffic dataset contains images with mostly small objects and tiny bounding boxes so even though the training result is relatively acceptable with a large number of epochs, we still do not get a good test mAP result. This is compounded by the extreme training time this YOLO model necessitates. It would take on the order of a week to train the YOLO model on a new dataset making its generalizability quite poor.

In addition to the original YOLO structure, we adapt ResNet and VGG to optimize performance. Adapting both improves the performance and ResNet is a better fit compared to VGG. For different model, we need to tune the parameters differently. Since YOLO is a deep learning algorithm, we found it hard to interpret the networks and hence it is difficult to tune and debug the models. We also experimented with image augmentation and found that it provided moderate if unspectacular results. While tuning and finding useful hyperparametrs was the first goal, we opted to simplify our resultant data collection.

Despite our poor results, real-time accurate object detection is a vital and ever growing need in our daily lives. The original YOLO algorithm, while computationally performant, performed acceptably to poorly on two new datasets. Now, improved versions of YOLO are available which optimizes both the performance and accuracy of the original YOLO far better than we were able to. It also improves YOLO's ability to detect small objects. We are confident that YOLO has its potential and will be developed further in the future.

Acknowledgements

Special Acknowledgement to Aladdin Persson who provided useful insight into implementing the YOLO algorithm with a series of in-depth YouTube videos.

References

[1] Redmon, Joseph, Divvala, Santosh, Girshick, Ross, and Farhadi, Ali, “You Only Look Once: Unified, Real-Time Object Detection,” in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 6/2016, vol. 2016-, pp. 779–788.

[2] Girshick, Ross, Donahue, Jeff, Darrell, Trevor, and Malik, Jitendra, “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation,” in 2014 IEEE Conference on Computer Vision and Pattern Recognition, 6/2014, pp. 580–587.

[3] Girshick, Ross, “Fast R-CNN,” in 2015 IEEE International Conference on Computer Vision (ICCV), 12/2015, vol. 2015, pp. 1440–1448.

[4] Shaoqing Ren, Kaiming He, Girshick, Ross, and Jian Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” IEEE transactions on pattern analysis and machine intelligence, vol. 39, no. 6, pp. 1137–1149, 2017.

[5] I. Kamp;K Technologies, “Cars and Traffic Signs Dataset: MakeML - Create Neural Network with ease,” MakeML. [Online]. Available: https://makeml.app/datasets/cars-and-traffic-signs. [Accessed: 11-Dec-2020].
[6] J. Nelson, “Mask Wearing Object Detection Dataset - raw,” Roboflow, 29-Nov-2020. [Online]. Available: https://public.roboflow.com/object-detection/mask-wearing/4. [Accessed: 11-Dec-2020].

[7] Everingham, Mark, et al. “The Pascal Visual Object Classes (VOC) Challenge.” International Journal of Computer Vision, vol. 88, no. 2, Springer Science and Business Media LLC, 2009, pp. 303–38, doi:10.1007/s11263-009-0275-4.

[8] Wu, Y., He, K. Group normalization. In Proceedings of the European conference on computer vision (ECCV), pp. 3-19, 2018.

[9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[10] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556, 2014.

[11] Redom, Joseph Chet. "Darknt: Open Source Neural Networks in C". [Online]. Available: "https://pjreddie.com/darknet/"

[12] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Kopf, Andreas and Yang, Edward and DeVito, Zachary and Raison, Martin and Tejani, Alykhan and Chilamkurthy, Sasank and Steiner, Benoit and Fang, Lu and Bai, Junjie and Chintala, Soumith, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 8024–8035, 2019

[13] Jia Deng, Wei Dong, Socher, Richard, Li-Jia Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248–255.

[14] Persson, Aladdin. “Pytorch YOLO From Scratch". [Online]. Available: "https://www.youtube.com/watch?v=n9_XyCGr-MI"