YOLO

Irfan Nafi

April 2020

1 Introduction

YOLO (You Only Look Once) is one of the most popular real-time object recognition algorithms in the field of computer vision. In this lecture, we will look at how YOLO works compared to more conventional models and how those differences make it faster. As of the moment, there are four versions of YOLO (YOLOv4 just came out!), with each version having several clever and complex improvements. To keep this lecture simple, we will cover the general theory, with additional focus on YOLOv3, instead of going too deep into one specific version.

2 Applications

Due to YOLO's ability to produce fast and accurate detections, it is applicable across a wide variety of areas, from the military to self-driving cars to everyday DIY projects. As of the moment, it is one of the most accurate computer vision algorithms for its speed, with YOLOv3 reaching mAPs (mean average precision - a metric to measure the accuracy of a model) of 61 at 20fps on the COCO dataset. There are also smaller versions of YOLO, called Tiny YOLOs, which can achieve FPS of over 240; however, this is at the cost of mAP with YOLOv3-Tiny reaching mAPs of only 33. YOLO is suited anywhere real-time object detection is needed, even being deployable on mobile applications due to Tiny YOLO's less complex architecture.

3 Process

Previous computer vision algorithms like Faster RCNN first propose regions for objects and then classify them, taking much longer. In the first few steps alone, it applies several convolutional layers, beginning with a layer that shrinks the image by 16 times. Then it applies two more convolutional layers to that feature map (the output of a filter applied to the previous layer). YOLO takes a different approach, instead of classifying and locating the object (through a bounding box) separately, YOLO does it simultaneously.

To find the dimensions of a bounding box, it might make sense to calculate the coordinates directly, however during training, this leads to unstable gradients, which makes it hard to train. Instead, YOLO takes advantage of the fact that, in real life, the same types of objects have similar shapes and aspect ratios with their dimensions, as shown by the image below.

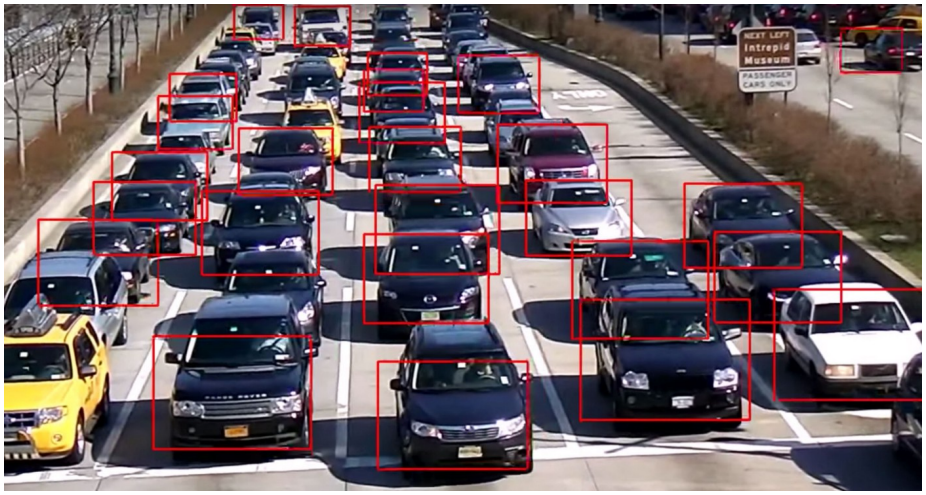

Figure 1. Bounding boxes are not arbitrary. Cars have very similar shapes with an approximate aspect ratio of 0.41.

This pattern across classes is why YOLO produces predetermined boxes called anchor boxes, or priors, per object to calculate the final bounding box as offsets of the anchor boxes (we will get into how it calculates the offsets later). These boxes are based on the trends of shape, size, and location of the class, as shown in Figure 1, and not on the actual object. This may seem a bit arbitrary, but the better the anchor boxes, the better the final bounding box.

To refine the anchor boxes to better cover the actual object, we use K-means clustering. K-means clustering is a method to extract the structure of data by partitioning it into K, non-overlapping clusters. Initially, we make random points within the data, and then through an iterative process, the points or centroids (because they are at the center of each cluster) positions are optimized. In YOLO, boxes are used instead of points.

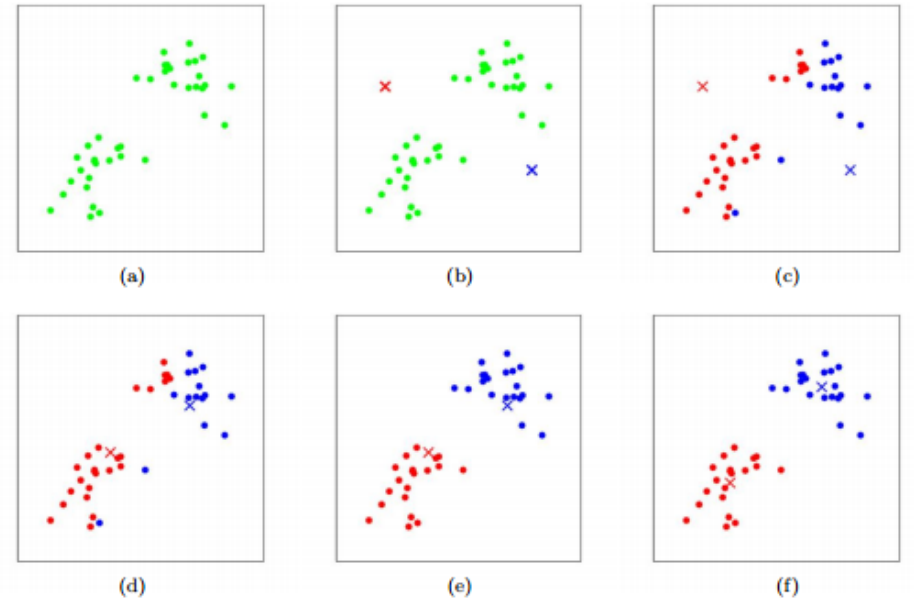

Figure 2. An example of how a K-means clustering algorithm iteratively optimizes the positions of the centroids for each cluster group.

To optimize the positions of these centroids, there needs to be a function or distance metric to tell you how much to offset the points. The easiest way may be to use the Euclidean distance. However, this results in larger boxes generating more error, so instead, we use IoU. We calculate IoU by dividing the intersection of the predicted boundary box and the ground truth (the actual pre-determined boundary of the object) over the union of the two boxes. If they don't intersect at all, then the IoU is just 0.

Figure 3. A diagram depicting how IoU is calculated.

Finally, using the anchor boxes, the width and height of the bounding boxes are calculated as offsets from the centroid of the anchor boxes. We will talk about the final calculation of the values later.

Let's bring together what we just covered and summarize the process from the beginning. YOLOv3 first splits up the image into S by S cells. For each grid cell, YOLO predicts five anchor boxes and takes the max across each cell for which class, if the most likely. Figure 4 illustrates this.

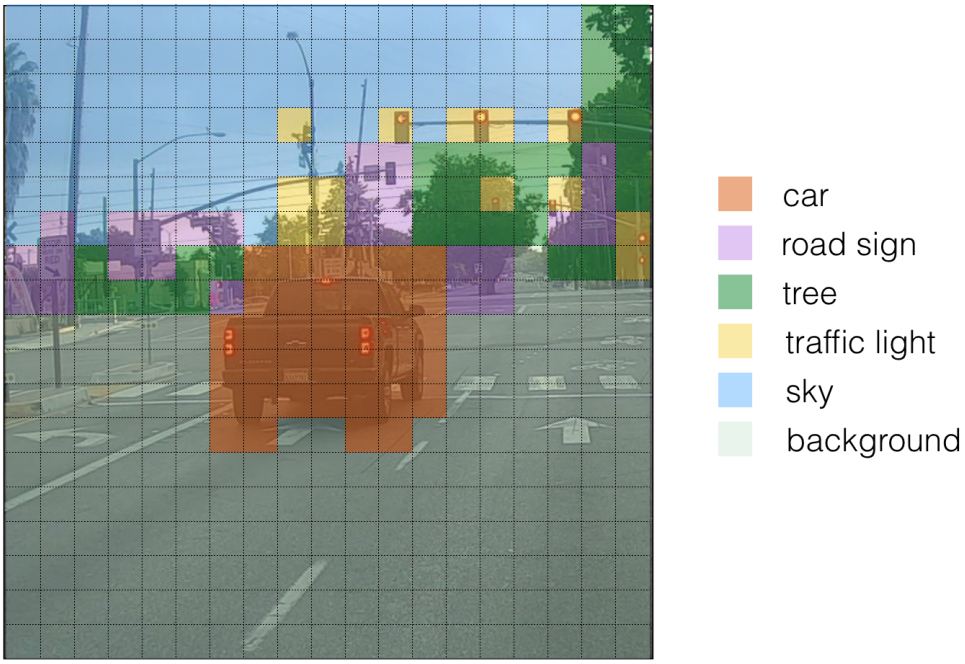

Figure 4. Diagram illustrating how YOLO takes he max of each cell to find the most likely class at that area.

Transformations applied to the anchor boxes result in 3 bounding box predictions per cell. This also limits how close objects can be for YOLO to detect them. The second image in Figure 7 illustrates all these class predictions, with the same classes having the same colors. Calculations for the final values are as follows:

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$

The final values, b_x, b_y, b_w and b_h, which are the center coordinates, width, and height, respectively. The network, itself, predicts 4 of these same types of coordinates, t_x, t_y, t_w and t_h with c_x and c_y being the top-left coordinates of the current cell. Finally, p_w and p_y are the width and height of the

anchor boxes in that cell. To calculate the center coordinates, we put these values through a sigmoid function and add the coordinates of the top-left of the cell.

Figure 5. An illustration of how anchor boxes are used to find the coordinates of the bounding box using offsets.

The width and height of the bounding box, or the offsets from the centroid of the anchor boxes, are calculated relative to the top-left corner of the current cell. It is also predicted by applying a log-space transformation to the network predicted dimensions and multiplying them with an anchor.

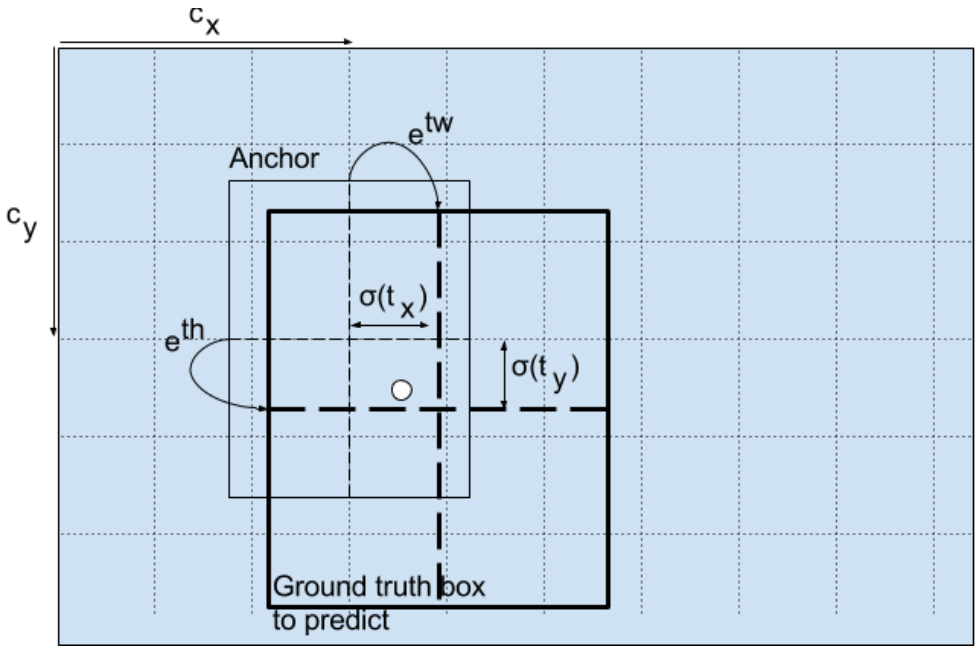

Figure 6. Diagram showing how the width and height of the bounding boxes are calculated based on the cell coordinates and anchor.

With each bounding box, there is also a score for the confidence of that box, or objectness score, that represents the probability that the bounding box contains the object. It is then passed through a sigmoid and interpreted as a probability. The reason we don't use softmax is because the probabilities from each possible class have to sum up to 1, so if there are two related classes associated with an object, say a woman and person, the probabilities will be low for both of them. If instead, a sigmoid is used, the probabilities for each class can be independent, so both woman and person can have high probabilities. Softmaxing assumes that the classes are mutually exclusive, which, in a lot of cases, they are not.

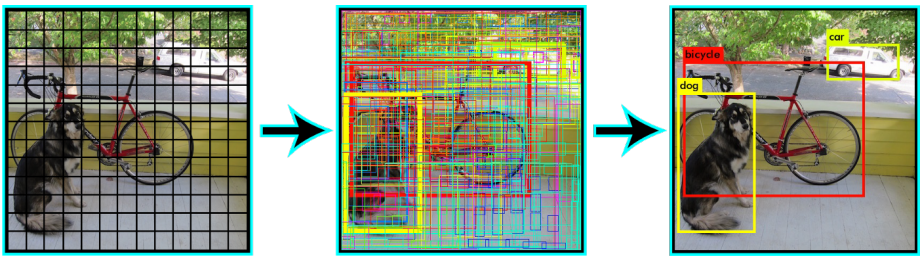

Figure 7. Diagram showing the 3 main steps YOLO takes to produce bounding boxes.

4 Architecture

4.1 Introduction

A computer vision architecture is the way the model processes an image, usually through a certain network of CNNs. YOLO is a fully convolutional network (FCN), meaning it only uses convolutional layers. The architecture behind YOLO is called darknet and is written in C and CUDA instead of TensorFlow. In YOLOv3 a new version of darknet is released, Darknet-53 which has 53 convolutional layers. YOLO's architecture is unique in that it uses the same network for both classification and localization of the object (through bounding boxes). The architecture is, thus, rather complex, so we will only look at its major features.

4.2 Darknet-53

As shown in figure 8, after each batch, there is a batch normalization layer, a method for improving the speed and stability of the network, with Leaky ReLu activation - similar to ReLu except there is a small negative slope for $x < 0$. Leaky ReLus help to solve the "dying ReLu" problem which occurs when ReLu updates a neuron in such a way so that it never activates on any data point again. This problem, alone, can kill up to 40% of the network. No form of pooling (a method to reduce the size of the feature map by taking the

average or max for a certain stride) is used, with a stride two convolutional layer down-sampling the feature maps.

	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 8. An architecture diagram for Darknet-53, alone.

YOLOv2 used Darknet-19, another custom deep architecture, which is a 19 layer network with an additional 11 layers for object detection. With only 30 layers, YOLOv2 struggled to classify smaller objects. Yolov3 uses a variation of Darknet-53 (which was trained on ImageNet - a dataset used as a metric for computer vision algorithms) with 53 more layers for a total of 106.

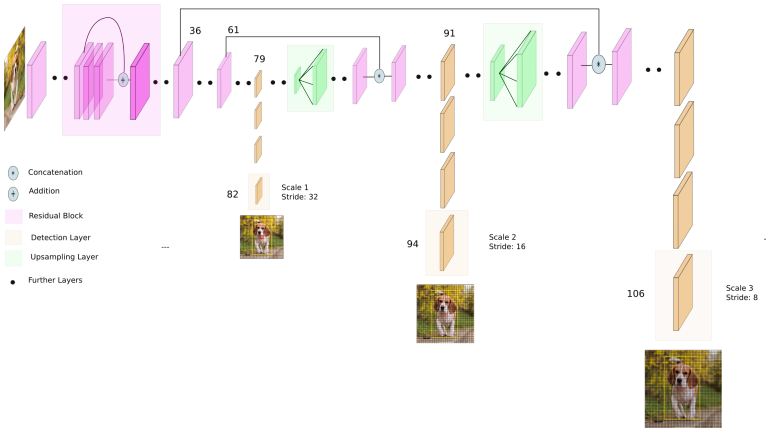

Figure 9. A network architecture diagram for YOLOv3, including the 53 additional layers on top of Darknet-53.

5 Overview on Training

5.1 Introduction

YOLO is relatively easy to implement in projects. If you're looking for general object detection, there are already weights like those trained on Microsoft's COCO (Common Objects in Context), which has 30 classes, and weights trained to classify 9,000 classes, a version called YOLO9000 (not very accurate). If you are looking to train YOLO, then a great place to start is Google's OIDv6 (Open Images Dataset), which not only has over 600 classes with bounding boxes, but also has object segmentation to train mask-based algorithms.

5.2 Managing Data

To train a specific class, you should have around 1-2 thousand images with a variety of lighting, object size, rotation, etc. If there isn't a sufficient number of files for training, then consider data augmentation. Data augmentation is a technique to modify existing images, through rotation, translation, Gaussian noise, etc. Data augmentation helps in reducing overfitting and improves generalization because of its ability to produce more training samples.

5.3 Steps

1. Split it into two distinct groups, one for training (70-90% of the data) and the other for testing
2. Download and build darknet to work with your machine, specifcally the GPUs
3. Annotations are human-made classifications of objects in images, which include bounding boxes and segmentations (the actual area of an object). You should only use datasets with annotations, but if there are not any available, use annotation software online.
4. Next, download a pre-trained model. This is called transfer learning and is what happens when a pre-trained model trained on other data is used to train another. Transfer learning is faster than starting from scratch, as it makes the learning process much smoother. For YOLOv3, darknet-53, which is trained on ImageNet, is a great place to start.
5. Choose all the hyperparameters and start training
6. As the training progresses, the log file will contain the loss from each batch. Once the loss has fallen below a certain threshold or stopped decreasing all together, then you should stop training.

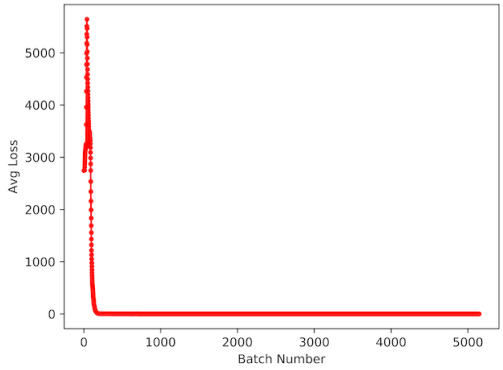

Figure 10. Example of a loss vs batch graph for a model trained on Darknet-53.

5.4 Advice

From my own experience, managing all the files (there are thousands per class!) was, at times, a tedious process that resulted in restarting training times. My advice is to start small, just to understand the process, and build-up towards the final number of classes. Also, make sure you configure your Makefile for the GPU you're training on. Alexey Bochkovskiy created a forked YOLO repository that is great for GPU compatibility on darknet and is, in general, a better version.

6 Conclusion

The field of computer vision has taken off in the last few years with the availability of large datasets and powerful computing. Every year, faster and more accurate algorithms are released, but none are as accurate as YOLO is at the speeds it is capable of. YOLO's ease of use makes it applicable to a wide range of areas, advancing the computer vision field as a whole. YOLO's different architecture and approach to processing an image, looking at it only once, is its differentiating factor and is what makes it so much faster.

7 References

The figures in this lecture were also from the following resources:

1. Redmon, J., and Farhadi, A. (2018). YOLOv3: An Incremental Improvement. Retrieved from https://pjreddie.com/media/files/papers/YOLOv3.pdf

2. Redmon, J., and Farhadi, A. (2017). YOLO9000: Better, Faster, Stronger. Retrieved from https://pjreddie.com/media/files/papers/YOLO9000.pdf

3. Redmon, Joseph, et al. "You Only Look Once: Unified, Real-Time Object Detection." ArXiv:1506.02640 [Cs], May 2016. arXiv.org, http://arxiv.org/abs/1506.02640.

4. YOLOv3 theory explained. (2019, July 4). Retrieved April 29, 2020, from https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

5. Hui, J. (2018, March 17). Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3. Retrieved April 29, 2020, from https://medium.com/@jonathan-hui/ real-time-object-detection-with-yolo-yolov2-28b1b93e2088

6. Tsang, S.-H. (2019, February 7). Review: YOLOv3 — You Only Look Once (Object Detection). Retrieved April 29, 2020, from https://towardsdatascience.com/ review-yolov3-you-only-look-once-object-detection-eab75d7a1ba6