

Do Thuan

**EVOLUTION OF YOLO ALGORITHM AND YOLOV5: THE STATE-OF-THE-ART  
OBJECT DETECTION ALGORITHM**

**EVOLUTION OF YOLO ALGORITHM AND YOLOV5: THE STATE-OF-THE-ART  
OBJECT DETECTION ALGORITHM**

Do Thuan  
Bachelor's Thesis  
DIN16SP  
Information Technology  
Oulu University of Applied Sciences

## ABSTRACT

Oulu University of Applied Sciences  
Bachelor's Degree in Information Technology

---

Author(s): Do Thuan

Title of the thesis: Evolution of YOLO Algorithm and YOLOv5: The State-of-the-art Object Detection Algorithm

Thesis examiner(s): Jaakko Kaski

Term and year of thesis completion: Spring 2021

Pages: 61

---

Object detection is one of the primary tasks in computer vision which consists of determining the location on the image where certain objects are present, as well as classifying those objects. In 2015, the YOLO (You Only Look Once) algorithm was born with a new approach, reframing object detection as a regression problem and performing in a single neural network. That made the object detection field explode and obtained much more remarkable achievements than just a decade ago. So far, combining with many of the most innovative ideas coming out of the computer vision research community, YOLO has been upgraded to five versions and assessed as one of the outstanding object detection algorithms. The 5th generation of YOLO, YOLOv5, is the latest version not developed by the original author of YOLO. However, the performance of the YOLOv5 is higher than the YOLOv4 in terms of both accuracy and speed.

This thesis investigates the most advanced inventions in the field of computer vision which were integrated into YOLOv5 as well as the previous versions. Using the Colab platform to implement object detection in the Global Wheat dataset contains 3432 wheat images. Subsequently, the YOLOv5 model will be evaluated and configured for improvement based on the results.

---

Keyword: Machine Learning, Artificial Intelligent, Python, Pytorch

# CONTENTS

1	INTRODUCTION .....	6
2	YOLO – YOU ONLY LOOK ONCE .....	8
2.1	Concepts .....	8
2.2	YOLOv1 architecture .....	11
2.3	Loss function .....	13
3	EVOLUTIONARY HIGHLIGHTS .....	15
3.1	YOLOv2 .....	15
3.1.1	Add Batch Normalization .....	15
3.1.2	High Resolution classifier .....	16
3.1.3	Convolutional with anchor box .....	16
3.2	YOLOv3 .....	19
3.2.1	Bigger network with ResNet .....	19
3.2.2	Multi-scale detector .....	21
3.3	YOLOv4 .....	24
3.3.1	Object detection architecture .....	24
3.3.2	Backbone – CSPDarknet53 .....	26
3.3.3	Neck – Additional block – SPP block .....	28
3.3.4	Neck – Feature Aggregation – PANet .....	30
3.3.5	Head – YOLOv3 .....	33
3.3.6	Bag of Freebies .....	34
3.3.7	Bag of Specials .....	35
4	5 <sup>TH</sup> GENERATION OF YOLO .....	36
4.1	Overview of YOLOv5 .....	36
4.2	Notable differences – Adaptive anchor boxes .....	37
5	IMPLEMENTING YOLOV5 ALGORITHM .....	38
5.1	Global Wheat Head detection dataset .....	38
5.2	Environment .....	39
5.3	Preparing the dataset for training .....	41
5.3.1	Creating the label text files .....	44
5.3.2	Splitting data into training set and validation set .....	46

5.3.3	Creating the data.yaml file .....	48
5.4	Training phase.....	50
5.4.1	Preparing the architecture.....	50
5.4.2	Training model .....	51
5.5	Inference with trained weight.....	55
6	CONCLUSION.....	59
7	REFERENCES .....	60

# 1 INTRODUCTION

People glancing at an image, can instantly recognize what the objects are and where they are located within the image. The ability to detect objects fast combined with the knowledge of a person helps to make an accurate judgment about the nature of the object. A system that simulates the ability of the human visual system to detect objects is something that scientists are researching on. Fast and accurate are the two prerequisites for which an object detection algorithm is examined.

Object detection is one of the classical problems in computer vision. It not only classifies the object in image but also localizes that object. In previous decades, the methods used to address this problem consisted of two stages: (1) extract different areas in the image using sliding windows of different sizes and (2) apply the classification problem to determine what class the objects belong to. These approaches have the disadvantage of demanding a large amount of computation and being broken down into multiple stages. That makes the system difficult to be optimized in terms of speed.

In 2015, researcher Joseph Redmon and colleagues introduced an object detection system that performs all the essential stages to detect an object using a single neural network for the first time, YOLO algorithm (the term as You Only Look Once). It reframes the object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. This unified model predicts simultaneously multiple bounding boxes and class probabilities for those objects covered by boxes. At the time of its release, the YOLO algorithm has produced impressive specifications that outstand the premier algorithms in terms of both speed and accuracy for detecting and determining object coordinates (Redmon, et al., 2016).

Over the next 5 years, the YOLO algorithm was upgraded to five versions (including the original version) with many of the most innovative ideas coming out of the computer vision research community. The first three versions are researched and developed by the author of the YOLO algorithm, Joseph Redmon. However, he announced to discontinue his research in the computer vision field after the release of YOLOv3. In early 2020, the official YOLO Github account released the YOLO update version 4, YOLOv4, published by Alexey Bochkovskiy, the Russian developer who built the first 3 versions of YOLO based on Joseph Redmon's Darknet framework. A month after the launch of YOLOv4, researcher Glenn Jocher and his Ultralytics LLC research department, who built YOLO

algorithms on the Pytorch framework, published YOLOv5 with a few differences and improvements. Although not developed by the team of algorithm authors, YOLOv5 has impressed with outstanding performance compared to all four previous versions (Jocher, 2020).

## 2 YOLO – YOU ONLY LOOK ONCE

YOLO came on the computer vision scene with a paper released in 2015 by Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection,” and immediately got a lot of attention from fellow computer vision researchers. Convolutional Neural Networks (CNN) such as Region-Convolutional Network (R-CNN) used Regions Proposal Networks (RPNs) before YOLO was invented, it produces proposal bounding boxes on the input image first, then runs a classifier on the bounding boxes and then apply post-processing to remove duplicate detections and refine the bounding boxes. It was not suitable for training individual stages of the R-CNN network separately. The R-CNN network was both difficult and sluggish to optimize.

The author's motivation is to build a unified model of all phases on a neural network. With the input image containing (or not) the objects, after passing forward through a single neural network of multiple convolutional networks, the system produces predictive vectors corresponding to each object appearing in the image. Instead of iterating the process of classifying different regions on the image, the YOLO system computes all the features of the image and makes predictions for all objects at the same time. That is the idea of "You Only Look Once". (Redmon, et al., 2016)

### 2.1 Concepts

The main idea of YOLOv1 is to apply a grid cell with the size of  $S \times S$  ( $7 \times 7$  default) into an image. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object (*Figure 1*). Therefore, all other cells disregard even that appearance of object revealed in multiple cells.

In order to implement object detection, each grid cell predicts  $B$  bounding boxes with their parameters and confidence scores for those boxes (*Figure 1*) (V Thatte, 2020). These confidence score reflects the presence or absence of an object in bounding box. The confidence score is defined as:

$$\text{confidence score} = p(\text{Object}) * IOU_{pred}^{truth}$$

with  $p(\text{Object})$  is the probability that there is an object inside the cell and  $IOU_{pred}^{truth}$  is intersection over union of prediction box and ground truth box.  $p(\text{Object})$  is in range 0-1, so the confidence score is close to 0 if no object exists in that cell. Otherwise, the score equal to  $IOU_{pred}^{truth}$ .



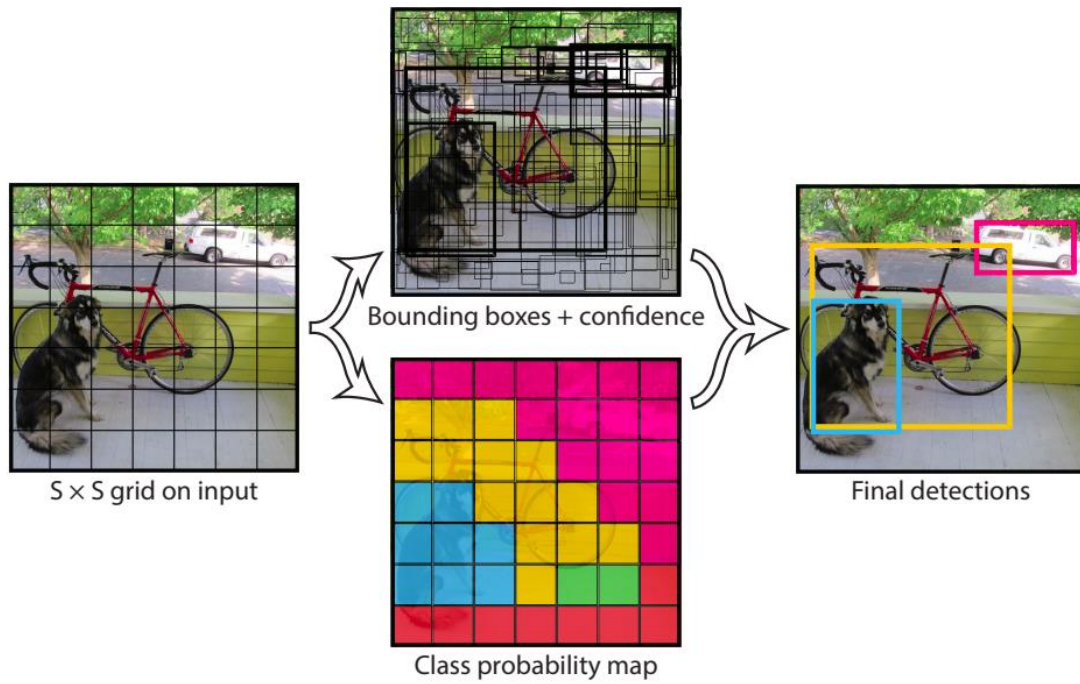


Figure 1. YOLO model with 7x7 grid cell was applied on input image. (Redmon, et al., 2016)

Besides, each bounding box consists of 4 other parameters  $(x, y, w, h)$  corresponding to  $(center\ coordinate(x, y), width, height)$  of a bounding box (Figure 2). Combining with the confidence score, each bounding box consist of 5 parameters.

(0, 0)

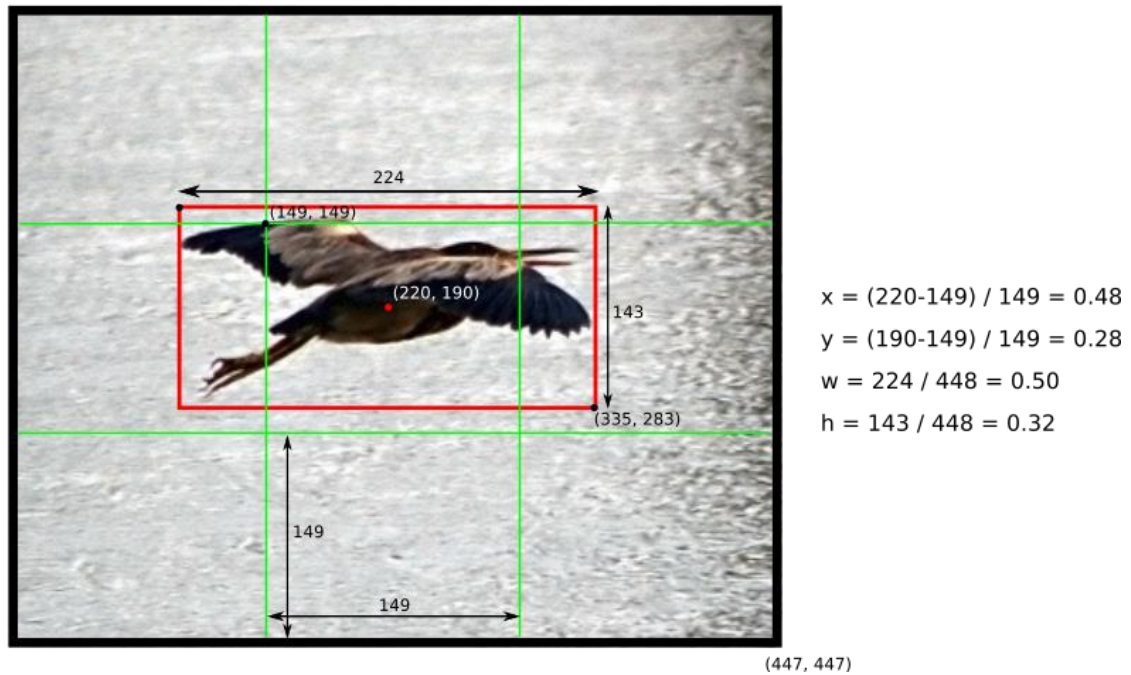
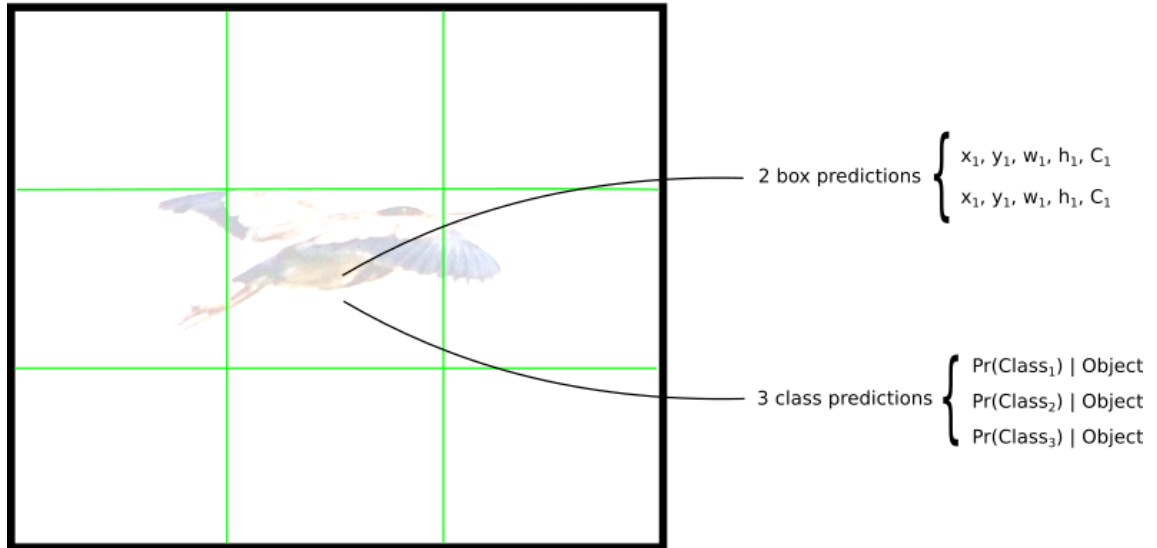


Figure 2. Parameters for a bounding box in 3x3 grid cell. (Menegaz, 2018)

The probability of an object predicted for each class in a grid cell denoted  $p(Class_i|Object)$ . Probability values for the  $C$  class will produce  $C$  output for each grid cell. The  $B$  bounding box of the same grid cell shares a common set of predictions about the object's class, meaning that all bounding boxes in the same grid cell have the same class. As shown in *Figure 3*, for example that there are 2 prediction boxes in center cell. They have different parameters  $(x, y, w, h, confidence\ score)$ . However, they have same 3 prediction classes.



*Figure 3. The bounding boxes of same grid cell share the set of prediction class. (Menegaz, 2018)*

Thus, the model has  $S \times S$  grid cells for an image. Each cell predicts  $B$  bounding boxes which consist of 5 parameters and share prediction probabilities of  $C$  classes. The total YOLO output of model parameters will be  $S \times S \times (5 * B + C)$  (Menegaz, 2018). For example, evaluating the YOLO model on famous COCO dataset which contains 80 classes and set each cell predicts 2 bounding boxes, the total output parameters are  $7 \times 7 \times (5 * 2 + 80)$ .

The purpose of the YOLO algorithm is to detect an object by precisely predicting the bounding box containing that object and localize the object based on the bounding box coordinates. Therefore, predicted bounding box vectors correspond to output vector  $\hat{y}$  and ground truth bounding box vectors correspond to vector label  $y$ . Vector label  $y$  and predicted vector  $\hat{y}$  can be indicated as *Figure 4* where the purple cell does not have any object, the confidence score of bounding boxes in purple cell equal to 0, then all remain parameters will be ignored.

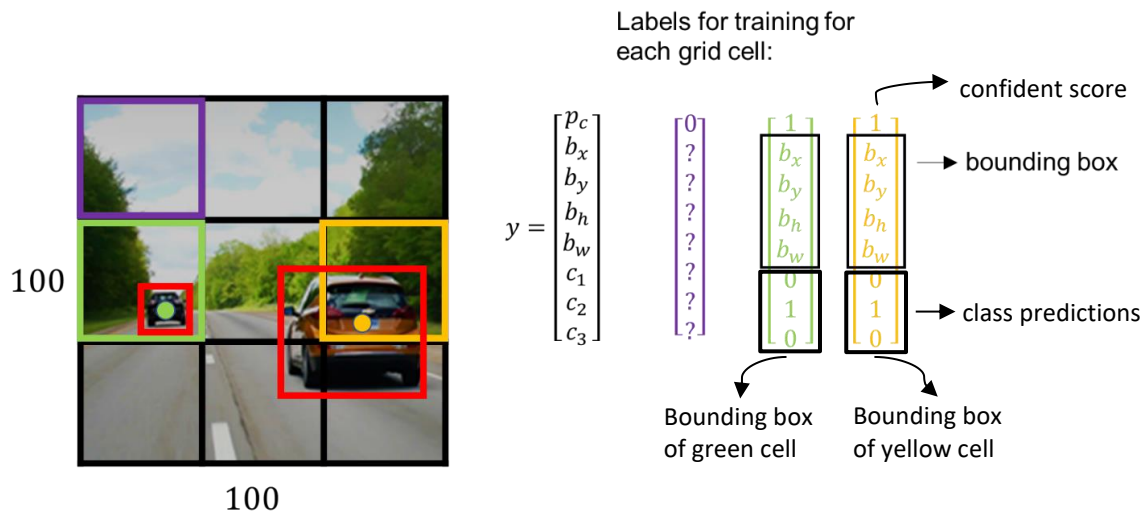


Figure 4. Specifying label vector  $y$  in a YOLO model has  $3 \times 3$  grid cells and predict object for 3 classes. (datahacker.rs, 2018)

Finally, YOLO applies Non-Maximum Suppression (NMS) to clean all bounding boxes which do not contain any object or contain the same object as other bounding boxes (Figure 1). By picking a threshold value, NMS removes all the overlap bounding boxes which have intersection over union (IOU) value higher than the threshold value. (ODSC Science, 2018)

## 2.2 YOLOv1 architecture

The YOLO model is designed to encompass an architecture that processes all image features (the authors called it Darknet architecture) and followed by 2 fully connected layers performing bounding box prediction for objects (Figure 5). This model was evaluated in the Pascal VOC dataset, where the authors used  $S = 7$ ,  $B = 2$  and  $C = 20$ . This explains why the final feature maps are  $7 \times 7$ , and the output size was  $(7 \times 7 \times (2 * 5 + 20))$ .

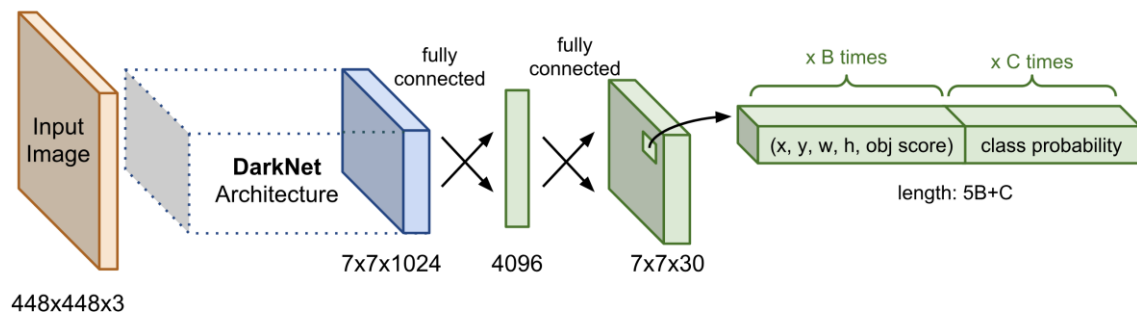


Figure 5. Preliminary YOLOv1 architecture.

The authors have introduced the fast-YOLO model with 9 CNN layers in Darknet architecture for uncomplicated datasets, and the normal-YOLO model with 24 CNN layers in Darknet architecture can deal with more complex datasets producing higher accuracy (Figure 6). The sequences of 1x1 and 3x3 convolutional layers were inspired by the GoogLeNet (Inception) model which helps reduce the features space from preceding layers (Menegaz, 2018). The final layer uses a Linear activation function instead of Leaky Rectified Linear Unit (leaky ReLU) activation as all other layers:

$$\phi(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases}$$

Name	Filters	Output Dimension
Conv 1	7 x 7 x 64, stride=2	224 x 224 x 64
Max Pool 1	2 x 2, stride=2	112 x 112 x 64
Conv 2	3 x 3 x 192	112 x 112 x 192
Max Pool 2	2 x 2, stride=2	56 x 56 x 192
Conv 3	1 x 1 x 128	56 x 56 x 128
Conv 4	3 x 3 x 256	56 x 56 x 256
Conv 5	1 x 1 x 256	56 x 56 x 256
Conv 6	1 x 1 x 512	56 x 56 x 512
Max Pool 3	2 x 2, stride=2	28 x 28 x 512
Conv 7	1 x 1 x 256	28 x 28 x 256
Conv 8	3 x 3 x 512	28 x 28 x 512
Conv 9	1 x 1 x 256	28 x 28 x 256
Conv 10	3 x 3 x 512	28 x 28 x 512
Conv 11	1 x 1 x 256	28 x 28 x 256
Conv 12	3 x 3 x 512	28 x 28 x 512
Conv 13	1 x 1 x 256	28 x 28 x 256
Conv 14	3 x 3 x 512	28 x 28 x 512
Conv 15	1 x 1 x 512	28 x 28 x 512
Conv 16	3 x 3 x 1024	28 x 28 x 1024
Max Pool 4	2 x 2, stride=2	14 x 14 x 1024
Conv 17	1 x 1 x 512	14 x 14 x 512
Conv 18	3 x 3 x 1024	14 x 14 x 1024
Conv 19	1 x 1 x 512	14 x 14 x 512
Conv 20	3 x 3 x 1024	14 x 14 x 1024
Conv 21	3 x 3 x 1024	14 x 14 x 1024
Conv 22	3 x 3 x 1024, stride=2	7 x 7 x 1024
Conv 23	3 x 3 x 1024	7 x 7 x 1024
Conv 24	3 x 3 x 1024	7 x 7 x 1024
FC 1	-	4096
FC 2	-	7 x 7 x 30 (1470)

Figure 6. Normal-YOLOv1 neural network model with 24 CNN layers and 2 fully connected layers

## 2.3 Loss function

The sum-squared error is the backbone of YOLO's loss function. There are multiple grid cells that do not contain any objects whose confidence score is zero. They overwhelm the gradients of cells that contain the objects. To avoid such overwhelming leading to training divergence and model instability, YOLO conducts the highest penalty for predictions from bounding boxes containing objects ( $\lambda_{coord} = 5$ ) and the lowest for predictions when no object is present ( $\lambda_{noobj} = 0.5$ ) (V Thatte, 2020). YOLO's loss function is calculated by taking the sum of all bounding box parameters' loss function including ( $x, y, w, h, confidence\ score, class\ probability$ ).

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

The first part of equation computes the loss related to the predicted bounding box position and ground truth bounding box position based on coordinates ( $x_{center}, y_{center}$ ).  $\mathbb{I}_{ij}^{obj}$  is define as 1 if object present inside  $j^{th}$  predicted bounding box in  $i^{th}$  cell, and 0 for otherwise. The predicted bounding box will be "responsible" for predicting an object based on which prediction has the highest current IOU with the ground truth.

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{I}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$

The second part of YOLO loss function calculates the error in prediction of bounding box width and height similar to first part of equation. However, the magnitude of error in the large boxes affect the equation less than in the small boxes. As both width and height are normalized in between 0 and 1, their square roots increase the differences for smaller values more than the larger values. Hence, the square root of the bounding box width and height is used instead of the width and height directly.

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{I}_{ij}^{obj} \left[ (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right]$$

The loss of confidence score is computed by both cases whether the object is present in the bounding box or not. The loss function only penalizes object confidence error if that predictor is responsible for the ground truth box.  $\mathbb{I}_{ij}^{obj}$  is equal to 1 when there is an object in the cell, and 0 otherwise.  $\mathbb{I}_{ij}^{noobj}$  is the opposite.

$$\sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{I}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{I}_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

The last part of the loss function is similar to the normal classification loss which computes the loss of class probability, except for the  $\mathbb{I}_{ij}^{obj}$  term. This term is used because YOLO does not penalize classification errors even when there are no objects present in the cell.

$$\sum_{i=0}^{S^2} \mathbb{I}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

### 3 EVOLUTIONARY HIGHLIGHTS

There have been five versions of YOLO published (including the original version) until now. Each version has been upgraded and integrated with the most advanced ideas coming out of the computer vision research community. Besides, some ideas have been also removed due to the inability to reach the required performance and accuracy of the algorithm. That makes YOLO even more powerful as one of today's top object detection algorithms. Before applying the state-of-the-art YOLOv5 model, discovering the remarkable improvements in the four previous versions is necessary for understanding the YOLOv5 algorithm.

#### 3.1 YOLOv2

##### 3.1.1 Add Batch Normalization

Batch normalization is one of the most popular methods of normalization in the deep learning model. It allows faster and more stable training of deep neural networks by stabilizing the distribution of the input layers during training (Ioffe, et al., 2015). The goal of this approach is to normalize the features (the output of each layer after passing activation) to a **zero-mean** state with a standard deviation of 1.

$$\text{Mini - batch mean: } \mu = \frac{1}{m} \sum_{i=1}^m z^{(i)}$$

$$\text{Mini - batch variance: } \sigma^2 = \frac{1}{m} \sum_{i=1}^m (z^{(i)} - \mu)^2$$

$$\text{Normalize: } z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\text{Scale and shift: } \tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$$

By applying batch normalization followed by all YOLOv2's convolution layers. This technique not only reduces training time but also increases the generalization of the network. In YOLOv2, batch normalization increased mAP (mean average precision) by about 2% (Redmon, et al., 2016). The network also does not need to use additional Dropouts to avoid overfitting.

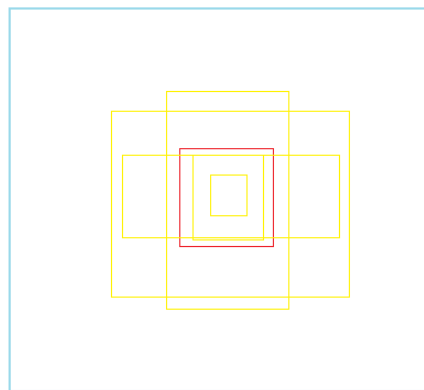
### 3.1.2 High Resolution classifier

In the original YOLO (YOLOv1), the first 20 convolution layers (in YOLOv1 architecture) were used to train feature extractor (classification network) with  $224 \times 224$  input image. Then the remaining 4 convolution layers and 2 fully connected layers were added, the resolution of the input image was simultaneously increased to  $448 \times 448$  to be used as object detector. (Kamal, 2019)

Whereas at YOLOv2, after completing the training phase of the feature extractor with the  $224 \times 224$  input image, the model continued to train the feature extractor for more 10 epochs with  $448 \times 448$  input image before using the architecture for training object detector. That help the model “adapt” to a large resolution of  $448 \times 448$  instead of suddenly increasing the image size when the feature extractor training phase moves to the object detector training phase. This high-resolution classification network gives an increase of almost 4% mAP.

### 3.1.3 Convolutional with anchor box

The idea of YOLOv1 is to use a grid cell to be responsible for detecting an object which has the center inside that grid cell. So, when two or more objects which have the center inside the same grid cell, the prediction may be flawed. To solve this problem, the author tried to allow a grid cell to predict more than one object. In YOLOv2, the author introduced an anchor box architecture to predict bounding boxes instead of using fully connected layers as in YOLOv1 (Redmon, et al., 2016). Anchor box is a list of predefined boxes that best match the desired objects. The bounding boxes were not only predicted based on ground truth boxes but also predefined  $k$  anchor boxes.



*Figure 7. For each grid cell (red), model predict 5 bounding boxes based on 5 anchor boxes (yellow) with different shapes.*



Instead of manually picking out the best-fit anchor boxes, the k-means clustering algorithm was used on the training set bounding boxes (including all the ground truth boxes) to cluster the bounding boxes which have similar shapes, then plot the average IOU with the closest centroid (Figure 8). But instead of using Euclidean distance, the author used IOU between the bounding box and the centroid. With various choices for  $k$ ,  $k = 5$  gives a good tradeoff for recall vs. complexity of the model. As shown in Figure 8, the author experimented in VOC and COCO dataset, the right image shows the trade-off between recall and model complexity based on number of clusters ( $k$ ). The right image shows 5 centroids (be used as anchor boxes) in both datasets. (Redmon, et al., 2016)

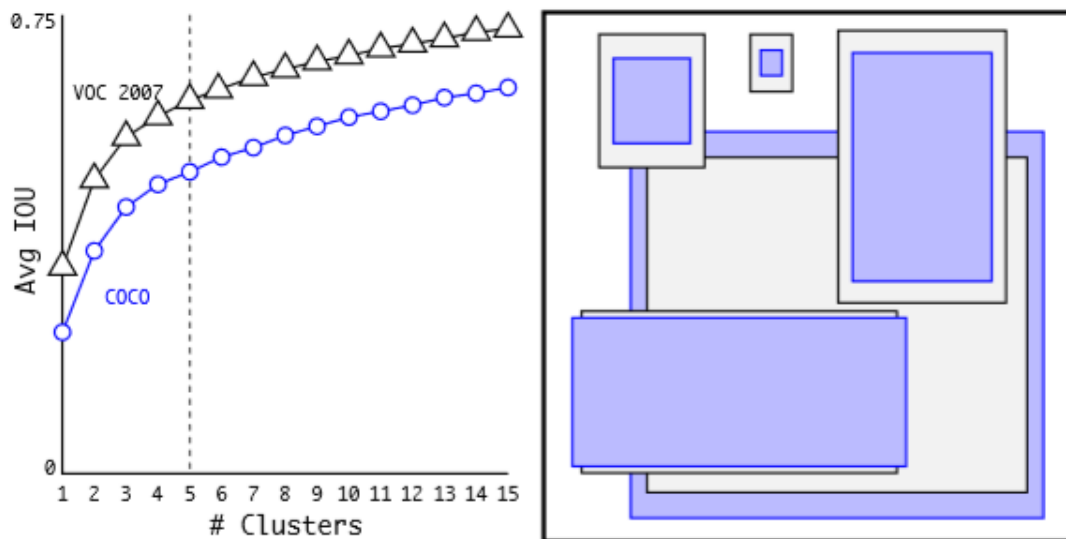
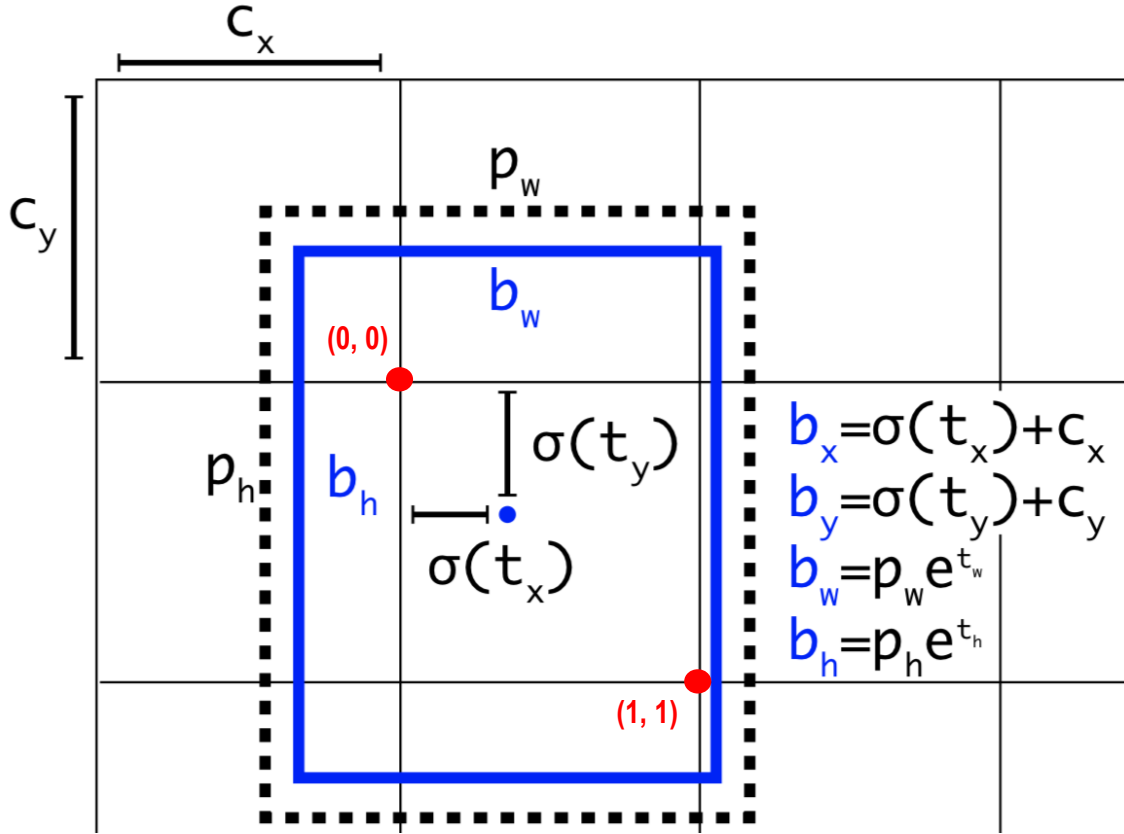


Figure 8. The k-mean clustering algorithm was used to choose best-fit anchor boxes. (Redmon, et al., 2016)

YOLOv1 has no restrictions on predicting the bounding position of box. When parameters are randomly initialized, the bounding box can be predicted anywhere in the image. This makes the model unstable in the early epochs of training. The position of bounding box can be distant from the grid cell responsible for predicting that bounding box.

Each grid cell in YOLO is specified in a scale of 0-1, the coordinate of the top-left point is (0, 0) and the bottom-right is (1, 1) as in Figure 9. Therefore, YOLOv2 used the sigmoid function ( $\sigma$ ) to restrict the value of the bounding box center in range 0-1, which in turn can set the bounding box predictions around the grid cell.

As shown in *Figure 9*, YOLOv2 assigns the bounding box (blue box) not only to the grid cell but also to one of the anchor boxes (dot box) which has the highest IOU with the ground truth box. The center coordinates of the box were predicted relative to the location of the filter application using a sigmoid ( $\sigma$ ) function. (Redmon, et al., 2016)



*Figure 9.* The sigmoid activation function was used to control the position of predicted bounding boxes. (Redmon, et al., 2016)

The model predicts 5 parameter values ( $t_x, t_y, t_w, t_h, t_o$ ) corresponding to (x, y, width, height, confidence score) for each bounding box as shown in the original YOLO. However, these parameters will be recalculated based on the correlation with the predefined anchor box (*Figure 9*). If the cell is offset from the top left corner of the image by ( $c_x, c_y$ ) and the given anchor box has width and height ( $p_w, p_h$ ), then the predictions were calculated as:

$$\begin{aligned}
 b_x &= \sigma(t_x) + c_x \\
 b_y &= \sigma(t_y) + c_y \\
 b_w &= p_w e^{t_w} \\
 b_h &= p_h e^{t_h} \\
 \sigma(t_o) &= p(\text{object}) * IoU(b, \text{object})
 \end{aligned}$$

Therefore, the final predicted bounding box will be  $(b_x, b_y, b_w, b_h, \sigma(t_o))$ . Since the model constrains the location prediction, the parametrization is easier to learn, making the network more stable. YOLOv2 has improved by almost 5% mAp with anchor boxes. (Redmon, et al., 2016)

### 3.2 YOLOv3

#### 3.2.1 Bigger network with ResNet

YOLO v2 used a custom deep 30-convolutional layers for Darknet architecture, more than YOLOv1 11 layers. For deep neural networks, more layers mean more accuracy. However, the input image was downsampled when forwarding to deeper layers leading to losing fine-grained features. That is why YOLOv2 often struggled with small object detections. ResNet brought the idea of skip connections to help the activations to propagate through deeper layers without gradient vanishing (Figure 10) (He , et al., 2015).

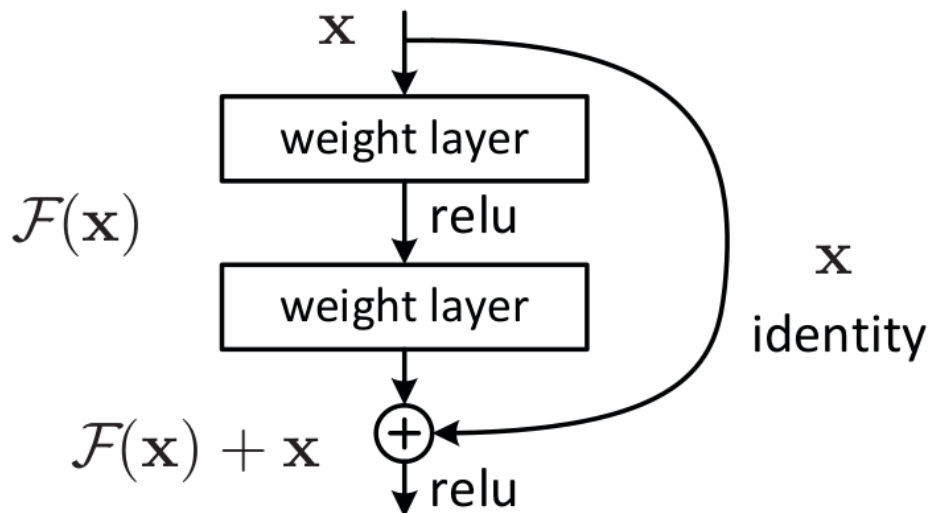


Figure 10. ResNet skip connection architecture.

YOLOv3 came up with a better architecture where the feature extractor used was a hybrid of YOLOv2, Darknet-53 (53 convolutional layers), and Residual networks (ResNet) (Redmon, et al., 2018). The network is built with the bottleneck structure ( $1 \times 1$  followed by  $3 \times 3$  convolution layers) inside each residual block plus a skip connection (Figure 11).

Thanks to residual blocks of ResNet, overlaying layers will not degrade network performance. Furthermore, the deeper layers get more information directly from the shallower layers, so it will not lose the mass of fine-grained features.

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

Figure 11. Darknet-53 architecture with 5 residual blocks (square box) containing bottle neck structure ( $1 \times 1$  followed by  $3 \times 3$  convolutional layers). (Yanjia LiYanjia Li, 2019)

The model used Darknet-53 architecture which originally has the 53-layer network for training feature extractor. After that, 53 more layers were stacked for the detection head for training object detector, making YOLOv3 a total of 106 layers fully convolutional underlying architecture.

### 3.2.2 Multi-scale detector

In 2 previous versions of YOLO, after training in the feature extractor with Darknet architecture, the input was forwarded to some more layers and finally make the predictions in the last layers of object detector. However, YOLOv3 appended the prediction layers aside network instead of stacking it at the last layers as before (Figure 12). The most notable feature of YOLOv3 is that it makes detections at 3 different scales (Redmon, et al., 2018). The features from the last 3 residual blocks were used for 3 different scale detectors.

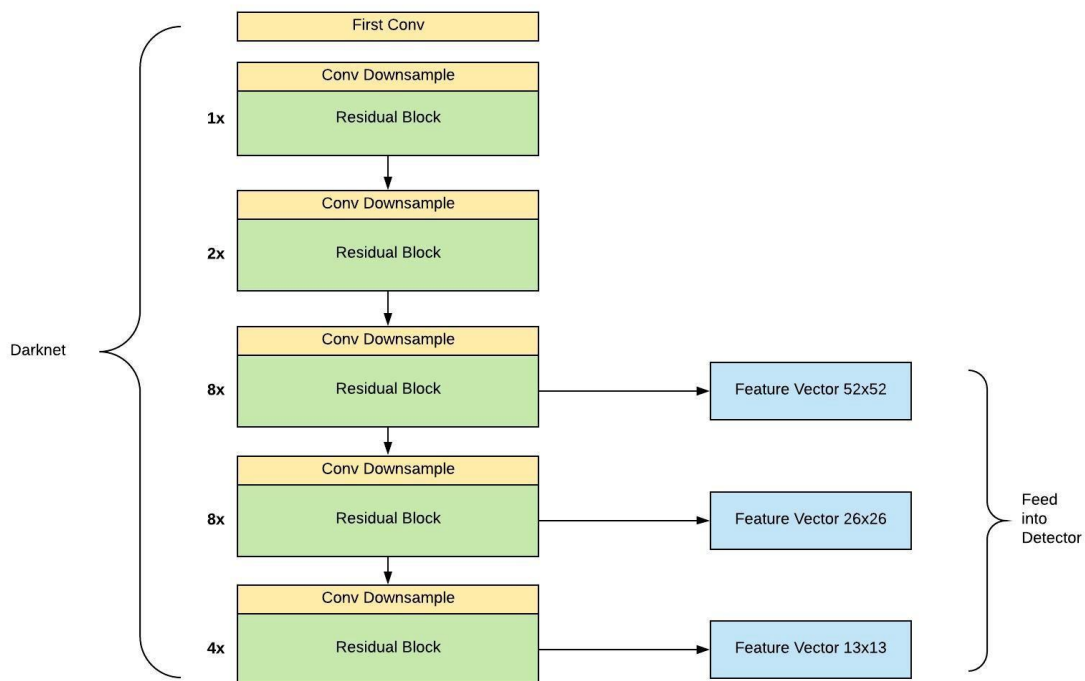


Figure 12. Multi-scale detector was appended aside network to make detection 3 times in 3 different scales. (Yanjia LiYanjia Li, 2019)

More specifically, YOLOv3 makes predictions at 3 scales in layers 82nd, 94th, and 106th, which are precisely given by stride of the network (or a layer is defined as the ratio by which it downsamples the input) are 32, 16, and 8, respectively (Figure 13).

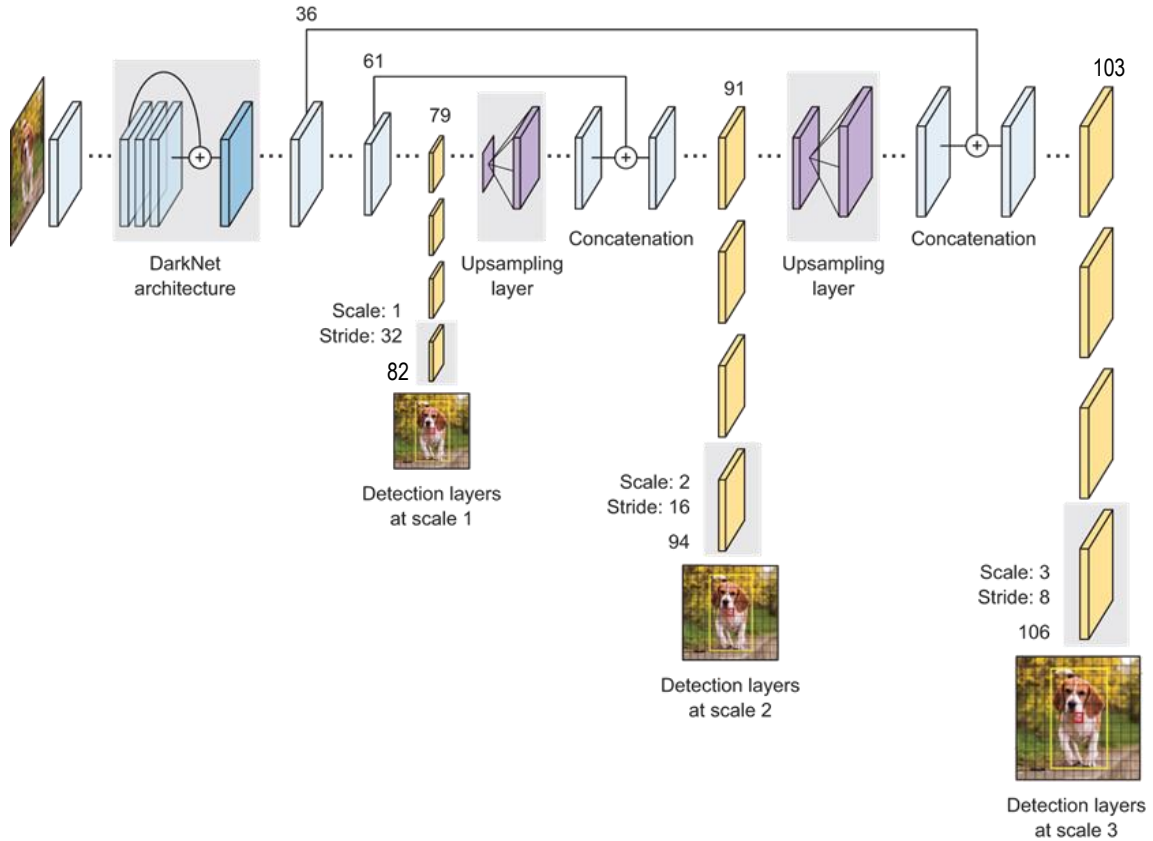


Figure 13. YOLOv3 network architecture combining both feature extractor and object detector. (Kathuria, 2018)

Unlike YOLOv1 where the bounding boxes were predicted by the same grid cell share a set of prediction probabilities of  $C$  classes, making each grid cell responsible for predicting only one object, the idea that a grid cell has the ability to predict multiple objects at the same time is initiated from YOLOv2, bounding boxes will detect different objects even if they are predicted by the same grid cell. Hence, predicted bounding boxes have their own set of prediction probabilities of  $C$  classes rather than share it together (Figure 14). The total YOLOv3's output parameters for each different detector will be  $S \times S \times (B \times (5 + C))$ .

The first detection is made by the 82nd layer. For ease of interpretation, the author used  $416 \times 416$  input image as default. After forwarding through the first 81 layers, the input image is downsampled by the stride of 32 and the resultant feature map would be of size  $13 \times 13$  corresponding to  $13 \times 13$  grid cells (Figure 13) (Kathuria, 2018). At each detection layer, the detection is done by applying  $1 \times 1$  detection kernels on feature maps. The  $1 \times 1$  kernel is responsible for predicting the  $B$  bounding box for each grid cell of the feature map. YOLOv3 was trained on COCO dataset with  $B = 3$  (3 bounding boxes for each cell) and  $C = 80$  (80 classes),

so the kernel size is  $1 \times 1 \times (3 \times (5 + 80)) = 1 \times 1 \times 255$ . At the first detection layer, the final resultant feature map would be  $13 \times 13 \times 255$  (Figure 14).

Image Grid. The Red Grid is responsible for detecting the dog

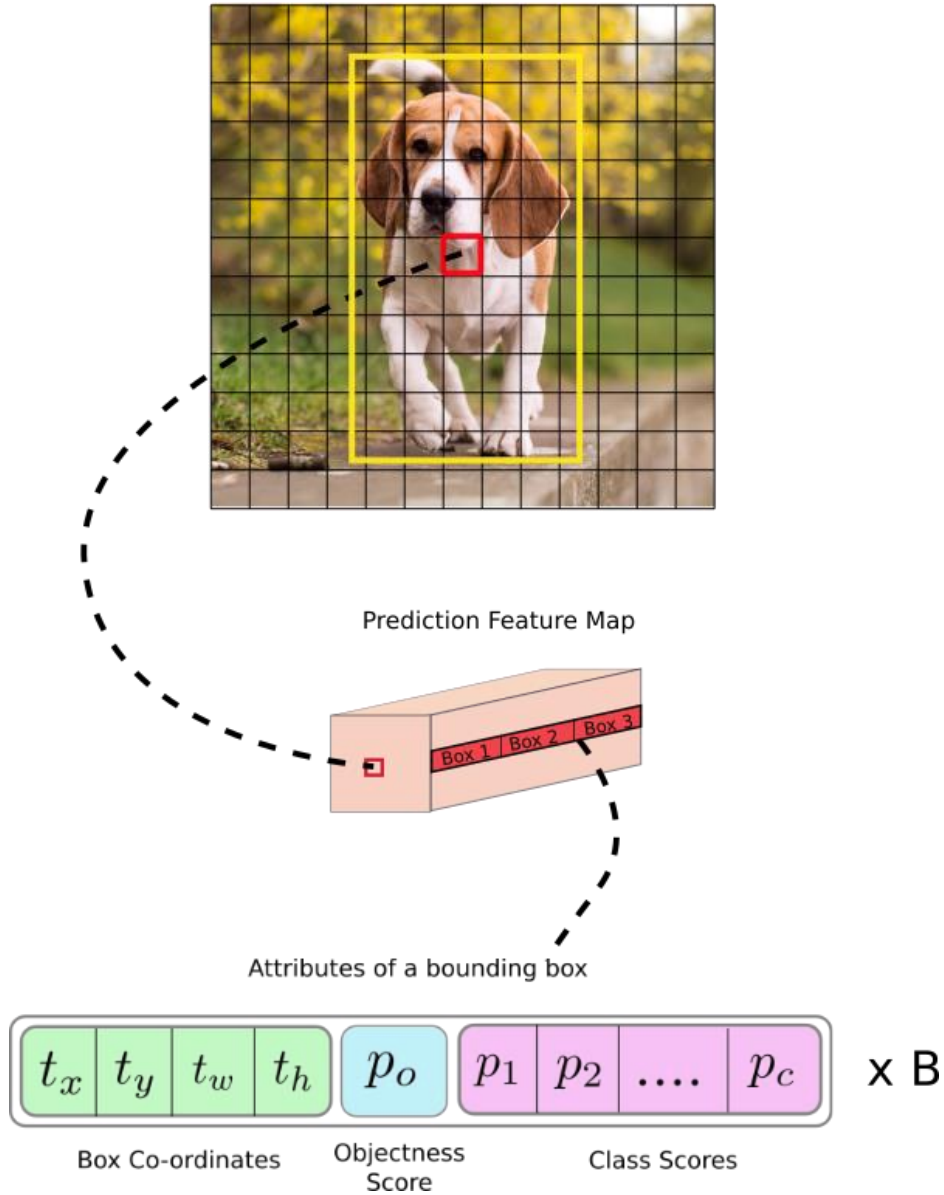


Figure 14. YOLOv3 detect an object by applying an  $1 \times 1$  kernel. (Kathuria, 2018)

The same procedure is repeated. However, before forwarding to 2 other detection layers for making predictions, the feature maps at layer 79th and 91st are upsampled. And after downsampling once again by the stride of 16 and 8, the feature maps have the size of  $26 \times 26$  and  $52 \times 52$  corresponding to the detection layer 94th and 106th (Figure 13). (Kathuria, 2018)

Moreover, detections at different scale layers help address the issue of detecting small objects, a frequent complaint with YOLOv2. The feature map with a larger size is more detailed. Thus, the large-scale detection layer ( $52 \times 52$ ) is responsible for detecting small objects, whereas the small-scale detection layer ( $13 \times 13$ ) detects larger objects.

By concatenates with the shallowed layers after upsampling in the deeper layer (concatenate with layer 61st before reach layer 91st and concatenate with layer 36 before reach layer 103rd as in *Figure 13*), the fine-grained features could be preserved from previous layers which help large-scale detection layer in detecting small objects.

### **3.3 YOLOv4**

The original YOLO algorithm was written by Joseph Redmon, who is also the author of a custom framework called Darknet. After 5 years of research and development to the 3rd generation of YOLO (YOLOv3), Joseph Redmon announced his withdrawal from the field of computer vision and discontinued developing the YOLO algorithm for concern of his research being abused in the military applications. However, he does not dispute the continuation of research by any individual or organization based on the early ideas of the YOLO algorithm.

In April 2020, Alexey Bochkovskiy, a Russian researcher and engineer who built the Darknet framework and 3 previous YOLO architecture on C-based on Joseph Redmon's theoretical ideas, has cooperated with Chien Yao and Hon-Yuan and published YOLOv4. (Bochkovskiy, 2020)

#### **3.3.1 Object detection architecture**

Along with the development of YOLO, many object detection algorithms with different approaches have achieved remarkable achievements as well. Since then, forming two concepts of architectural object detection: One-stage detector and Two-stage detector (*Figure 15*).



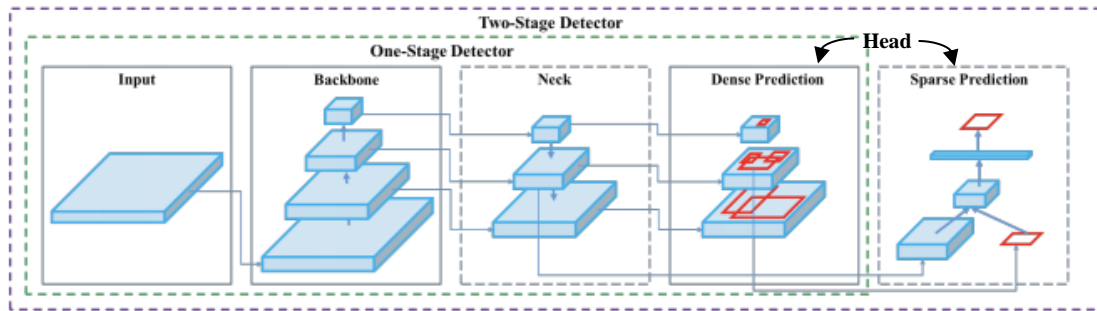


Figure 15. Two concepts of architectural object detection. (Solawetz, 2020)

The common point of all object detection architectures is that the input image features will be compressed down through feature extractor (Backbone) and then forwarding to object detector (including Detection Neck and Detection Head) as in Figure 15. Detection Neck (or Neck) works as a feature aggregation which is tasked to mix and combine the features formed in the Backbone to prepare for the detection step in Detection Head (or Head).

The difference appearing here that Head is responsible for making detection including localization and classification for each bounding box. The two-stage detector implements these 2 tasks separately and combines their results later (Sparse Detection), whereas the one-stage detector implements it at the same time (Dense Detection) as in Figure 15 (Solawetz, 2020). YOLO is a one-stage detector, therefore, You Only Look Once.

The YOLOv4 author performed a series of experiments with many of the most advanced innovation ideas of computer vision for each part of the architecture (Figure 16). (Bochkovskiy, et al., 2020)

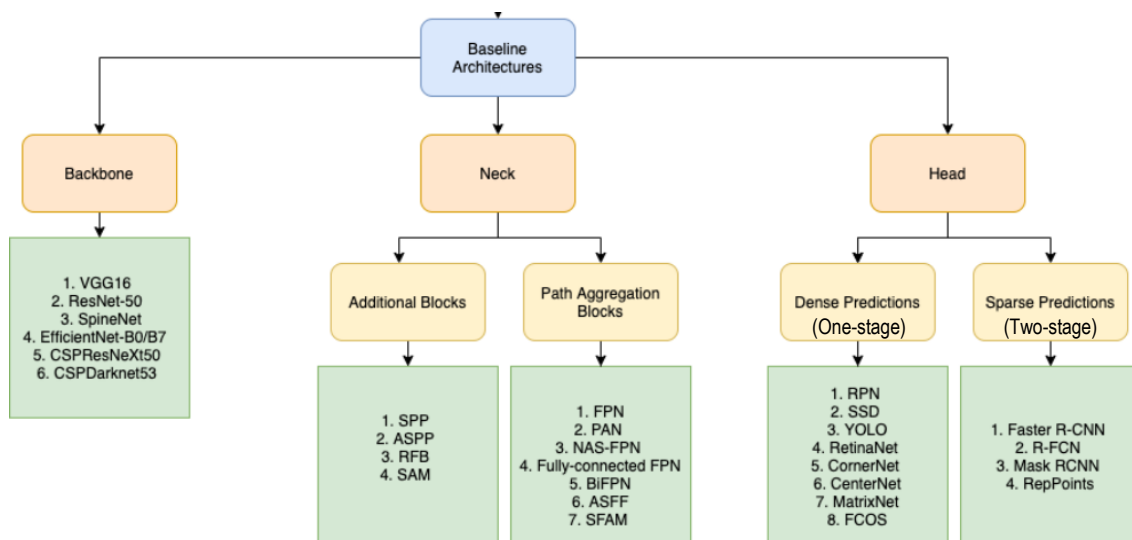


Figure 16. Diagrams of the most advanced innovation ideas applied by the author to each part of the YOLOv4 architecture.

### 3.3.2 Backbone – CSPDarknet53

The backbone (feature extractor) of the YOLOv4 model was considered by the authors among 3 options: CSPResNext53, CSPDarknet53 and EfficientNet-B3, the most advanced convolutional network at that time. Based on theoretical justification and lots of experiments, CSP Darknet53 neural network was determined to be the most optimal model (Figure 17).

Backbone model	Input network resolution	Receptive field size	Parameters	Average size of layer output (WxHxC)	BFLOPs (512x512 network resolution)	FPS (GPU RTX 2070)
CSPResNext50	512x512	425x425	20.6 M	<b>1058 K</b>	31 (15.5 FMA)	62
CSPDarknet53	512x512	725x725	<b>27.6 M</b>	950 K	52 (26.0 FMA)	<b>66</b>
EfficientNet-B3 (ours)	512x512	<b>1311x1311</b>	12.0 M	668 K	11 (5.5 FMA)	26

Figure 17. Comparison table of 3 backbone networks. (Huang, et al., 2018)

The CSPResNext50 and the CSPDarknet53 (CSP stands for Cross Stage Partial) are both derived from the DenseNet architecture which uses the previous input and concatenates it with the current input before moving into the dense layer (Huang, et al., 2018). DenseNet was designed to connect layers in a very deep neural network with the aim of alleviating vanishing gradient problems (as ResNet).

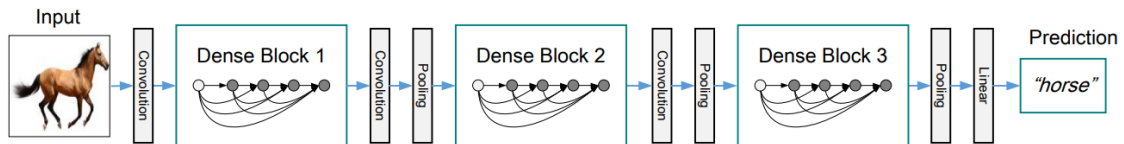


Figure 18. DenseNet architecture with 3 Dense blocks. The layers between 2 blocks referred to as transition layers. (Huang, et al., 2018)

To clarify, each stage of DenseNet consist of a dense block and a transition layer (Figure 18), each dense block is constructed of  $k$  dense layers. The input after goes through the dense block will come to the transition layer for changing size (downsample or upsample) via convolution and pooling (Figure 19). The output of the  $i^{th}$  dense layer will concatenate with its own input to form the input for the next  $(i + 1)^{th}$  layer. For example, at  $1^{st}$  dense layer, the input  $x_0$  after forward through convolutional layers has produced the output  $x_1$ . Then, the output  $x_1$  concatenate with its own input  $x_0$  and that concatenated outcome become the input of  $2^{nd}$  dense layer (Figure 19).

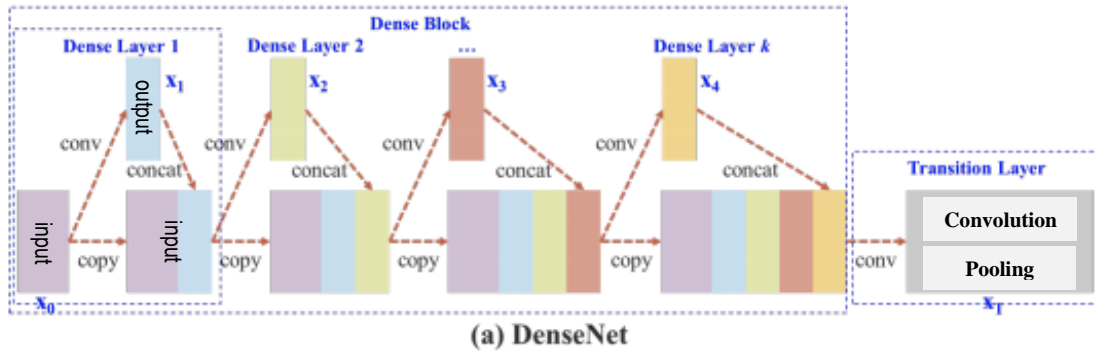


Figure 19. The process of the input processing in a dense block. (Wang, et al., 2019)

The CSP (Cross Stage Partial) is based on the same principle of the above DenseNet except that instead of using the full-size input feature map at the base layer, the input will be separated into 2 portions. A portion will be forwarded through the dense block as usual and another one will be sent straight on to the next stage without processed (Figure 20). This will result in different dense layers repeatedly learn copied gradient information. (Wang, et al., 2019)

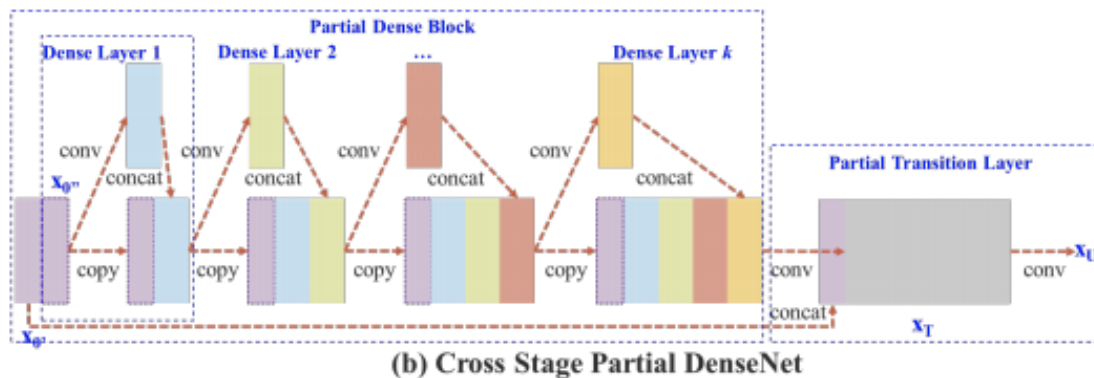


Figure 20. The process of the input processing in a partial dense block. (Wang, et al., 2019)

Combining these ideas with Darknet-53 architecture in YOLOv3, the residual blocks were replaced by the dense blocks. CSP maintains features through propagation, encourages the network to reuse features, and reduces the number of network parameters, helps to preserve fine-grained features for forwarding to deeper layers more efficiently. Considering that excessive increase of the densely connected convolutional layers may lead to a decrease in detection speed, only the last convolutional block which can extract the richer semantic features in the Darknet-53 backbone network is improved to be a dense block (Figure 21). (Huang, et al., 2019)

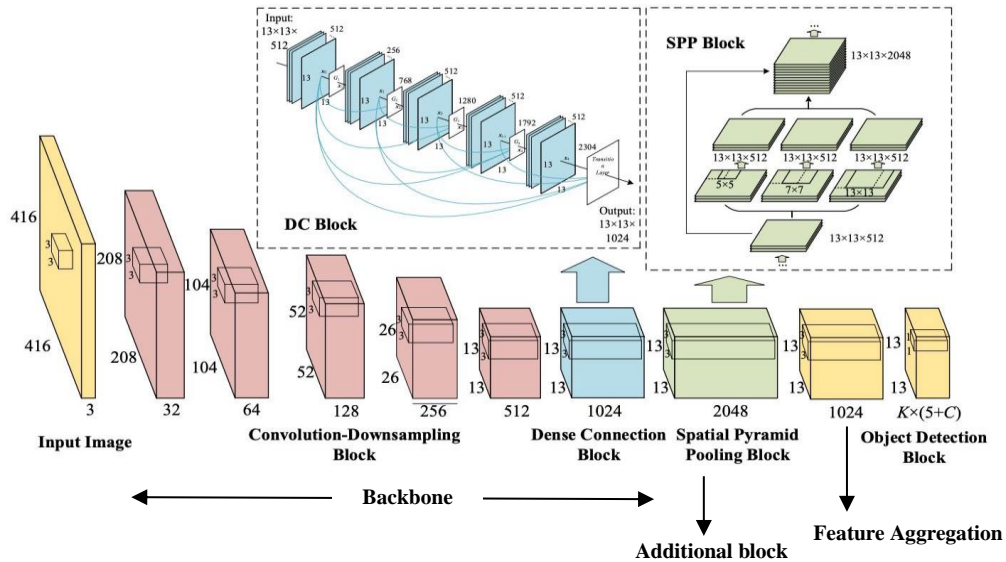


Figure 21. Dense block connection and Spatial Pyramid Pooling Based YOLOv2 architecture. (Shah, 2020)

### 3.3.3 Neck – Additional block – SPP block

Before forwarding to feature aggregation architecture in the neck, the output feature maps of the CSPDarknet53 backbone were sent to an additional block (Spatial Pyramid Pooling block) to increase the receptive field and separate out the most important features (Figure 21).

Many CNN-based (convolutional neural network) models contain fully connected layers which only accept input images of specific dimensions. SPP was born with the aim of generating a fixed-size output irrespective of the input size. Not only that, but SPP also helps to extract important features by pooling multi-scale versions of itself. As shown in Figure 22, the input feature maps have been duplicated to  $n$  versions ( $n = 3$  in this case) where each version was conducted max pooling with kernels of different sizes. By doing that, the SPP block simultaneously extracts  $n$  different types of important features. (He, et al., 2015)

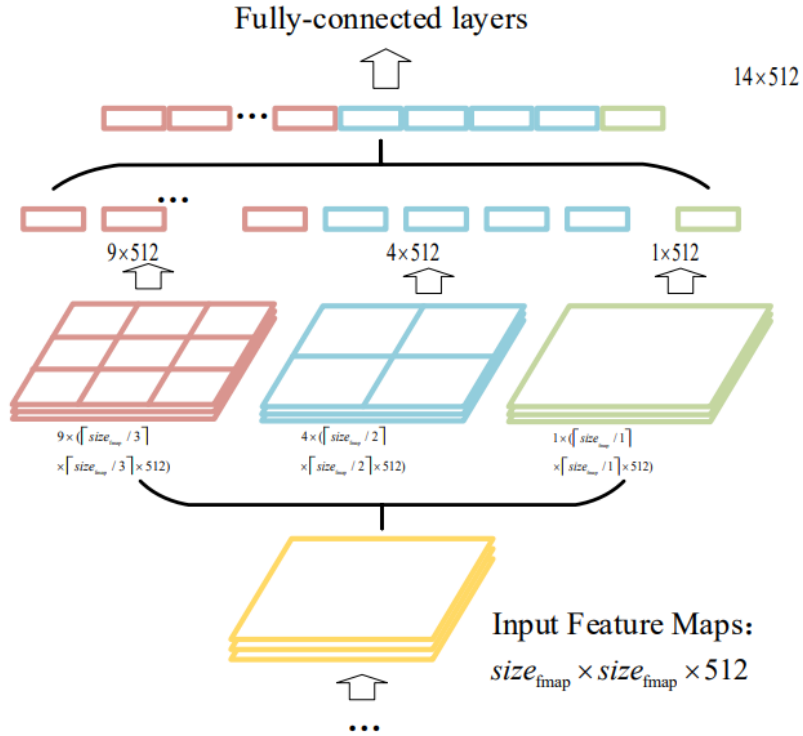


Figure 22. Classical SPP block. (Huang, et al., 2019)

Since fully connected layers were removed from YOLOv2, the YOLO algorithm has become an FCN-based (fully convolution network) model which allows the input images of different dimensions. Besides, YOLO has to make predictions and localizations about the coordinates of the bounding boxes based on the  $S \times S$  grid cell drawn on the image. Thus, converting two-dimensional feature maps into a fixed-size one-dimensional vector is not necessarily desirable.

For that reason, the SPP block has been modified to preserve the output spatial dimension (Figure 23). The new SPP block was located adjacent to the backbone (Figure 21). For reducing the number of input feature maps are sent to the SPP block (from 1024 to 512), a  $1 \times 1$  convolution is used between the backbone and the SPP block. After that, then input feature maps are duplicated and pooled in different scales based on the same principle of classical SPP block except that the padding is used to keep a constant size of the output feature maps, then 3 feature maps will retain the sizes of  $size_{fmap} \times size_{fmap} \times 512$ .

Different from the classical SPP block where the feature maps (Figure 22) were converted to a one-dimensional vector after conducting multi-scale max-pooling, the new SPP block (Figure 23) concatenates these 3 feature maps pooled with the sizes of  $size_{fmap} \times size_{fmap} \times 512$  and

including the input feature maps to avoid loss of important features in the case 3-scale max-pooling is not enough. Hence, the input not only extracted the important features that made the training easier but also kept the spatial dimension. (Wang, et al., 2019)

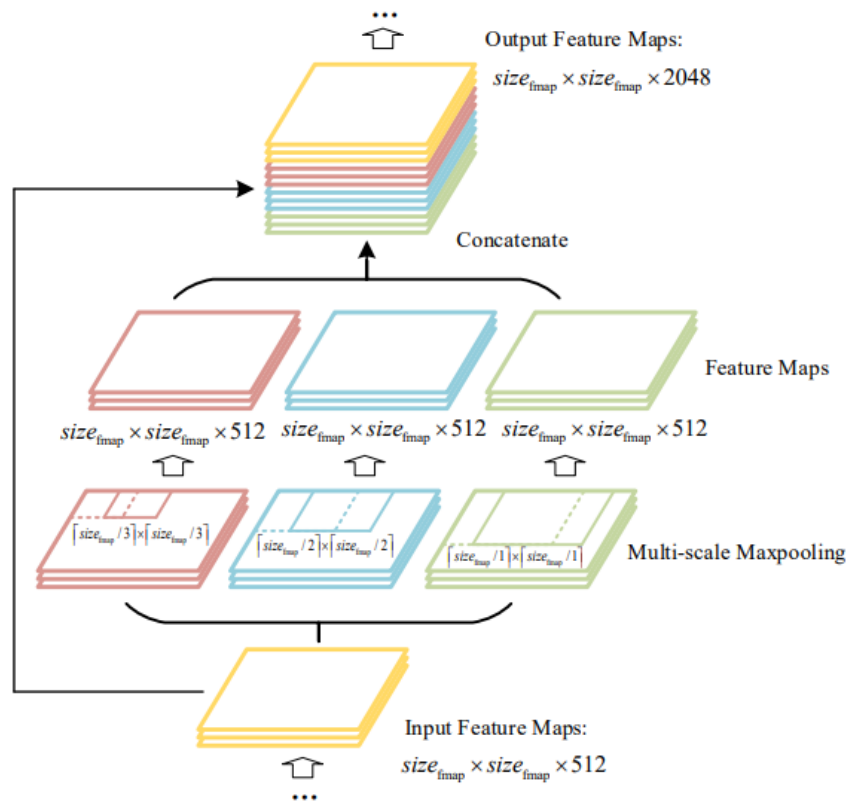


Figure 23. The new SPP block adapted to YOLO. (Huang, et al., 2019)

### 3.3.4 Neck – Feature Aggregation – PANet

The input image after forwarding through the backbone, the image features are processed into semantical features (or learned features). In other words, from the low-level layers, the deeper that the input image goes through, the complexity of semantical features will be more increased while the spatial resolution of feature maps will be more decreases due to downsampling. This leads to a loss of spatial information as well as fine-grained features. In order to preserve these fine-grained features, Joseph Redmon applied the idea of Feature Pyramid Network (FPN) architecture for YOLOv3's neck.

The FPN architecture implemented a top-down path to transfer the semantical features (from the high-level layer) and then concatenate them to fine-grained features (from the low-level layer in the backbone) for predicting small objects in the large-scale detector (Figure 24). (Hui, 2020)

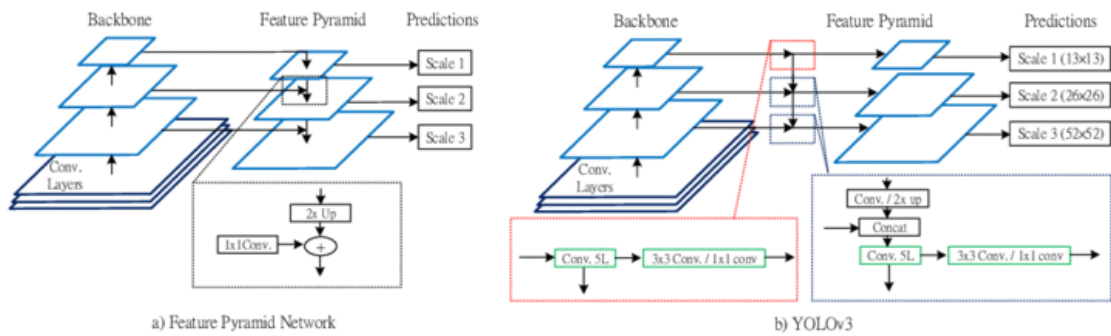


Figure 24. (a) The original FPN architecture. (b) The modified FPN architecture used in YOLOv3. (Gochoo, 2020)

Path Aggregation Network (PAN) is an advanced version of FPN. Because the flow in FPN architecture is the top-down path, hence only the large-scale detector from low-level layers in FPN is able to simultaneously receive the semantic features from high-level layers and fine-grained features from low-level layers in the lateral backbone (Figure 25a). Currently, the small-scale detector from high-level layers in FPN uses only semantic features for detecting objects. To improve the performance for the small and medium-scale detector, the idea of concatenating the semantic features and fine-grained features at high-level layers was considered.

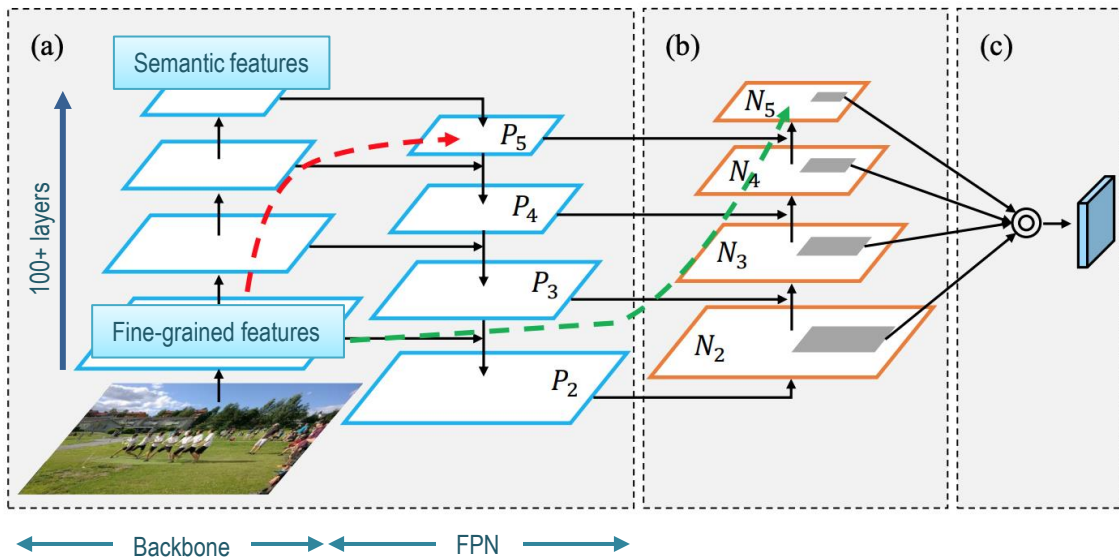


Figure 25. PANet architecture including (a) FPN backbone, (b) bottom-up path augmentation, (c) adaptive feature pooling. (Liu, et al., 2018)

However, for the deep neural network nowadays, the backbone of them contains lots of layers (can be more than 100 layers). Therefore, in FPN, the fine-grained features have to take a long path for traveling from low-level to high-level layers (the red path in Figure 25 or red path in Figure 26a).

The authors of the PAN architecture proposed to add a bottom-up augmentation path beside the top-down path used in FPN (Figure 25b). Thereby, a “shortcut” was created to directly connect fine-grained features from low-level layers to the top ones (green path in Figure 25 or green path in Figure 26b). This “shortcut” includes less than 10 layers, allowing ease of information flow (Liu, et al., 2018).

As shown in Figure 25, the backbone and FPN architecture are presented the same. But actually, the backbone contains lots of layers, the 4 drawn layers represent for the layers which were concatenated to multi-scale detectors in FPN architecture, not represent for the entire backbone. That the reason why the green path “shortcut” (Figure 25) goes through less than 10 layers although longer than the red path in the same picture.

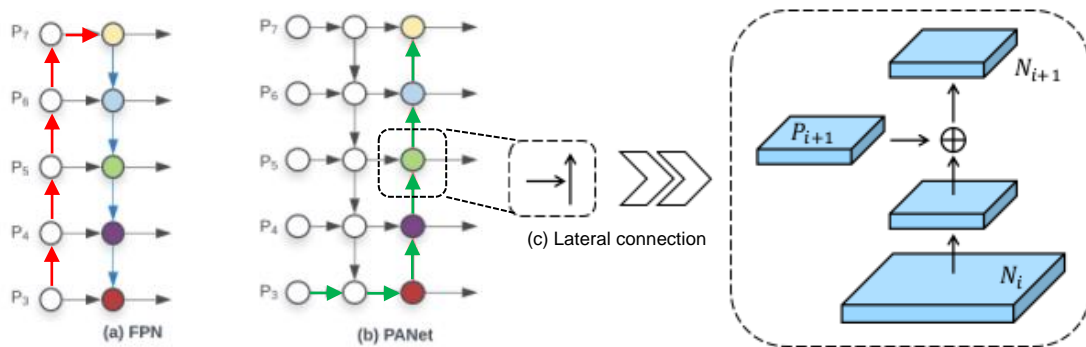


Figure 26. (a) FPN architecture. (b) PAN architecture. (c) Connection in bottom-up augmentation path. (Solawetz, 2020)

The bottom-up augmentation path can be witnessed as a replica of the FPN top-down path with each stage containing layers that produce feature maps with the same spatial sizes. These feature maps are connected to the lateral architecture by the element-wise addition operation (Figure 27a), whereas in modified PAN architecture for YOLOv4, the authors replaced it with the concatenation operation (Figure 27b). This helps the flow of information missing out on neither FPN features nor the bottom-up augmentation path features.



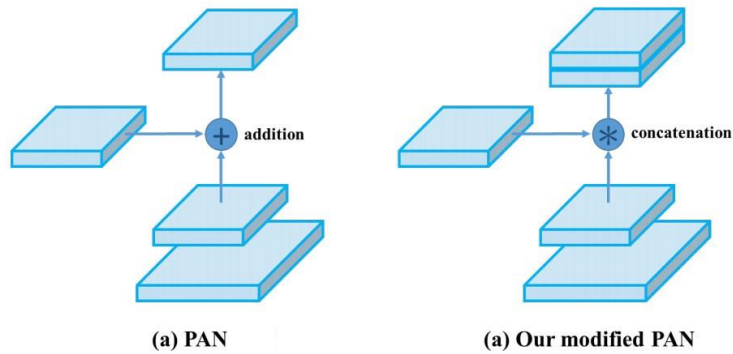


Figure 27. (a) Original PAN. (b) Modified PAN for YOLOv4. (Bochkovskiy, et al., 2020)

In FPN, the predictions were made separately and independently at different scale levels (Figure 24). This may produce duplicated predictions and not utilize information from other feature maps. PAN fused all the output feature maps of bottom-up augmentation pyramid by using ROI (Region of Interest) Align and fully connected layers with element-wise max operation (Figure 25c and Figure 28). So that, all the variabilities of feature maps are aggregated and used for predictions. (Liu, et al., 2018)

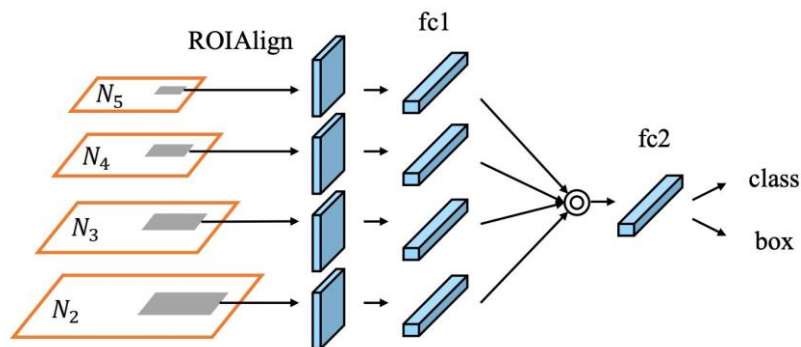


Figure 28. PAN used ROI Align for adaptive pooling and fully connected layers for fusing features from all stages. (Liu, et al., 2018)

### 3.3.5 Head – YOLOv3

In the case of a one-stage detector, the function of the head is to perform dense predictions. The dense prediction is the final prediction composed of a vector containing the predicted bounding box coordinates (center, height, width), the prediction confidence score, and the probability classes. YOLOv4 deploys the identical head as YOLOv3 for detection with the anchor-based detection steps, and three levels of detection granularity (Solawetz, 2020).

### 3.3.6 Bag of Freebies

The authors have experimented and applied a set of marginal optimization methods into YOLOv4 architecture with the aim of further improving algorithm performance and accuracy. These improvements are referred to by the authors by the term "Bag of Freebies" and "Bag of Specials". (Kanjee, 2020)

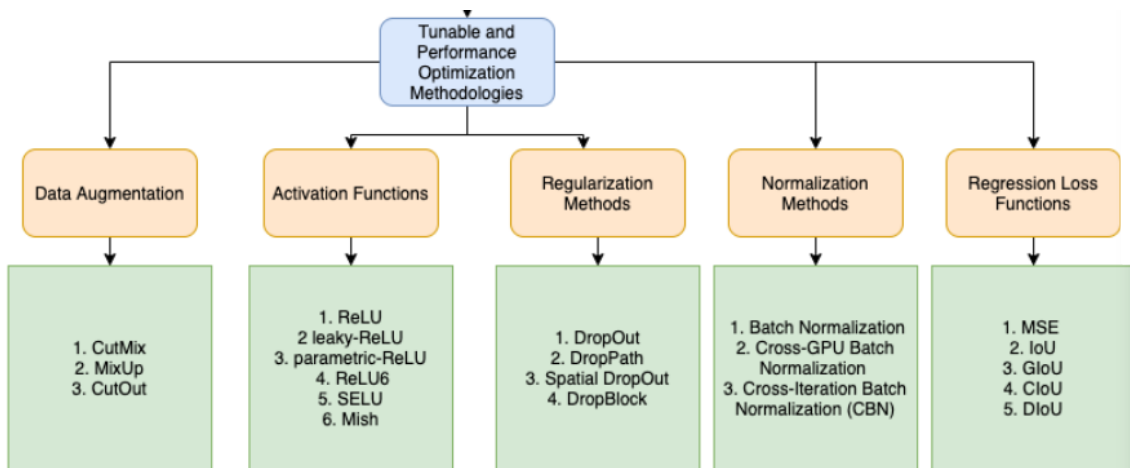


Figure 29. Multiple state-of-the-art optimization methods were experimented for each field.

"Bag" refers to a set of improvement methods and "Freebies" means that these improvements in this "Bag" can help to increase the performance and accuracy of the model without any cost to hardware. Hence, essentially the architecture will get more additional performance for free.

Through innumerable experiments and objective evaluation based on detailed statistics, the authors of YOLOv4 screened out and determined the following improvement methods for Bag of Freebies and applied them to YOLOv4.

**For backbone:** CutMix and Mosaic data augmentations, DropBlock regularization, Class label smoothing. (Bochkovskiy, et al., 2020)

**For detector (neck and head):** Complete Intersection over Union (CloU-loss), Cross-mini-Batch Norm (CmBN), DropBlock regularization, Mosaic data augmentation, Self-Adversarial Training (SAT), Eliminate grid sensitivity, using multiple anchors for a single ground truth, Cosine annealing scheduler, Optimal hyper-parameters, Random training shapes. (Bochkovskiy, et al., 2020)

### 3.3.7 Bag of Specials

Besides Bag of freebies, the authors introduced more another set of improvement method, called Bag of Specials. "Specials" here refers to getting something of value at a discount or for cheap. That means Bag of Specials includes the most advanced optimization methods which require the architecture to pay a small cost for significantly improving the performance accuracy of object detection.

**For backbone:** Mish activation, Cross-stage partial connections (CSP), Multi-input weighted residual connections (MiWRC). (Bochkovskiy, et al., 2020)

**For detector (neck and head):** Mish activation, Spatial Pyramid Pooling block (SPP-block), Spatial Attention Module (SAM-block), Path Aggregation Network (PANet), Distance Intersection over Union Non-Maximum Suppression (DIoU-NMS). (Bochkovskiy, et al., 2020)

## 4 5<sup>TH</sup> GENERATION OF YOLO

A month after YOLOv4 was released, researcher Glenn and his team published a new version of the YOLO family, called YOLOv5 (Jocher, 2020). Glenn Jocher is a researcher and CEO of Ultralytics LLC. YOLO models were developed on a custom framework Darknet which is written mainly in C by Alexey Bochkovsky. Ultralytics is the company that converts previous versions of YOLO on one of the most famous frameworks in the field of deep learning, PyTorch which is written in the Python language.

### 4.1 Overview of YOLOv5

Besides, Glenn Jocher is also the inventor of the Mosaic data augmentation and acknowledged by Alexey Bochkovsky in the YOLOv4 paper (Bochkovskiy, et al., 2020). However, his YOLOv5 model caused lots of controversy in the computer vision community because of its name and improvements.

Despite being released a month after YOLOv4, the start of research for YOLOv4 and YOLOv5 was quite close (March – April 2020). For avoiding collision, Glenn decided to name his version of YOLO, YOLOv5. Thus, basically, both researchers applied the state-of-the-art innovations in the field of computer vision at that time. That makes the architecture of YOLOv4 and YOLOv5 very similar and it makes many people dissatisfied with the name YOLOv5 (5th generation of YOLO) when it does not contain multiple outstanding improvements compared to the previous version YOLOv4. Besides, Glenn did not publish any paper for YOLOv5, causing more suspicions about YOLOv5.

However, YOLOv5 possessed the advantages in engineering. YOLOv5 is written in Python programming language instead of C as in previous versions. That makes installation and integration on IoT devices easier. In addition, the PyTorch community is also larger than the Darknet community, which means that PyTorch will receive more contributions and growth potential in the future. Due to being written in 2 different languages on 2 different frameworks, comparing the performance between YOLOv4 and YOLOv5 is difficult to be accurate. But after a while, YOLOv5

has proved higher performance than YOLOv4 under certain circumstances and partly gained confidence in the computer vision community besides YOLOv4.

## 4.2 Notable differences – Adaptive anchor boxes

As mentioned above, the YOLOv5 architecture has integrated the latest innovations similar to the YOLOv4 architecture, thus there are not many brilliant differences in theory. The author did not publish a detailed paper, but only launched a repository on Github and updates improvements there. By dissecting its structure code in file `.yaml`, the YOLOv5 model can be summarized as follows (Jocher, 2020):

- Backbone: Focus structure, CSP network
- Neck: SPP block, PANet
- Head: YOLOv3 head using GloU-loss

The remarkable point mentioned by the YOLOv5 author is an engineering difference. Joseph Redmon introduced the anchor box structure in YOLOv2 and a procedure for selecting anchor boxes of size and shape that closely resemble the ground truth bounding boxes in the training set. By using the k-mean clustering algorithm with different  $k$  values, the authors picked the 5 best-fit anchor boxes for the COCO dataset (containing 80 classes) and use them as the default. That reduces training time and increases the accuracy of the network.

However, when applying these 5 anchor boxes to a unique dataset (containing a class not belonged to 80 classes in the COCO dataset), these anchor boxes cannot quickly adapt to the ground truth bounding boxes of this unique dataset. For example, a giraffe dataset prefers the anchor boxes with the shape thin and higher than a square box. To address this problem, computer vision engineers usually run the k-mean clustering algorithm on the unique dataset to get the best-fit anchor boxes for the data first. Then, these parameters will be configured manually in the YOLO architecture.

Glenn Jocher proposed integrating the anchor box selection process into YOLOv5. As a result, the network has not to consider any of the datasets to be used as input, it will automatically "learning" the best anchor boxes for that dataset and use them during training. (Solawetz, 2020)

## 5 IMPLEMENTING YOLOV5 ALGORITHM

### 5.1 Global Wheat Head detection dataset

Wheat is the most cultivated cereal crop in the world, along with rice and maize. With high nutritional value, it is an ingredient used to process a wide variety of foods such as bread, cereals, etc. Its popularity as a food and crop makes wheat widely studied. During the cultivation process, farmers look at the wheat heads to evaluate health and maturity when making management decisions in their fields. But to control all the wheat on the entire enormous field manually is impossible. The development of a system that automatically detects wheat heads can give assistance to farmers to detect anomalies in their crops in time.

However, accurate wheat head detection in outdoor field images can be visually challenging. There is usually an overlap of dense wheat plants, and the wind can blur the photographs. Both make it difficult to identify single heads. Additionally, appearances vary due to maturity, color, genotype, and head orientation. Finally, because wheat is grown worldwide, different varieties, planting densities, patterns, and field conditions must be considered. April 2020, with contributions from 9 research institutes from 7 countries: The University of Tokyo, Institute national de recherche pour l'agriculture, l'alimentation et l'environnement, Arvalis, ETHZ, University of Saskatchewan, University of Queensland, Nanjing Agricultural University, and Rothamsted Research. The Global Wheat Head Detection (GWHD) was created contains 3,432 high-resolution RGB images and more than 140,000 labelled wheat heads collected with a wide range of genotypes. (E. David, 2020)



Figure 30. An example of wheat head was labelled with bounding boxes.

## 5.2 Environment

Google Colab (Google Collaboratory) is a free Integrated Development Environment (IDE) from Google to support research and learning about Artificial Intelligent (AI). Collaboratory provides a code environment as Jupyter Notebook, and it is free to use Graphic Process Unit (GPU) and Tensor Process Unit (TPU). Google Colab has pre-installed libraries that are very popular in Deep Learning research such as PyTorch, TensorFlow, Keras, and OpenCV.

Due to machine learning/deep learning algorithms require the system to have high speed and processing power (usually based on GPU), normal computers are not equipped with GPU. Therefore, Colab supplies GPU (Tesla V100) and TPU (TPUv2) on cloud, one of the highest performing GPUs at the moment, to give assistance to AI researchers.

Colab provides 25GB RAM and 150GB main disk. Checking GPU usage status:

```
[1] !nvidia-smi

Sun Jan 10 12:53:28 2021

+-----+
| NVIDIA-SMI 460.27.04    Driver Version: 418.67    CUDA Version: 10.1    |
+-----+-----+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M.         |
+-----+-----+-----+
|   0   Tesla V100-SXM2...    Off          | 00000000:00:04:0  Off   |          0          |
| N/A   33C    P0      24W / 300W |  0MiB / 16130MiB |           0%      Default |
|                                           MIG M.         | ERR!
+-----+-----+-----+

+-----+
| Processes:
| GPU  GI  CI           PID  Type  Process name                        GPU Memory
|   ID  ID  ID                                         Usage
+-----+-----+-----+
| No running processes found
+-----+
```

Figure 31. Colab provides Tesla V100-SXM2 GPU on cloud for training deep learning models.

As with previous versions, the YOLOv5 architecture was built theoretically based and released via a repository on GitHub. As mentioned, Ultralytic builds YOLOv5 on the PyTorch framework, one of the most popular frameworks in the AI community. However, this is only a preliminary architecture, researchers can configure the architecture to give the best results depending on their problems such as adding layers, removing blocks, integrating additional image process methods, changing the optimization methods or activation functions, etc.

```
!git clone https://github.com/ultralytics/yolov5 # clone repo
!pip install -qr yolov5/requirements.txt # install dependencies (ignore errors)
%cd yolov5

import torch # Pytorch
from IPython.display import Image, clear_output # to display images
from utils.google_utils import gdrive_download # to download models/datasets

clear_output()
print('Setup complete.')
```

Setup complete.

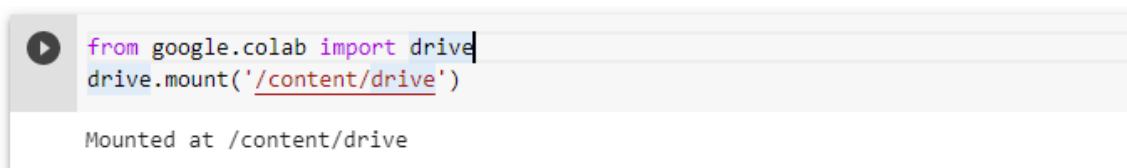
Figure 32. Cloning and installing the YOLOv5 repository.



### 5.3 Preparing the dataset for training

The Global Wheat Head Detection (GWHD) dataset has been published for use without license. Since Colab is a Google service, it allows linking to a personal Google Drive account to get data from Drive for use in training the model as well as save the results later.

After the GWHD dataset is downloaded to the personal computer (PC), it is uploaded to Google Drive and linked to Colab:

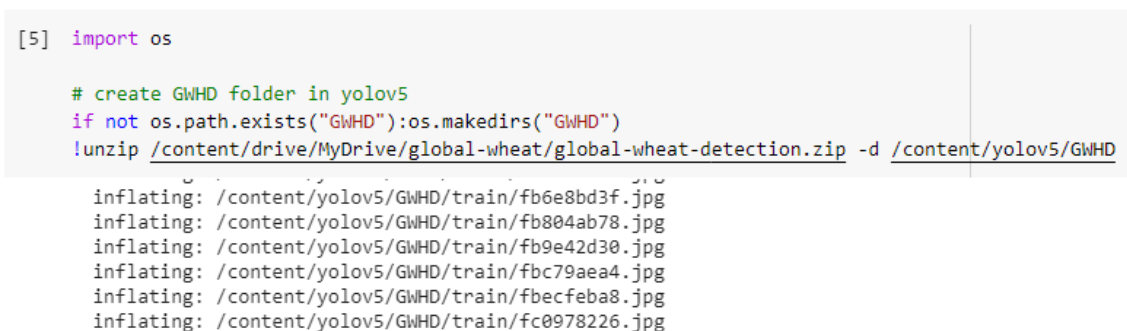


```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

Figure 33. Mounting to the Google Drive.

Unzip the dataset in Drive to the YOLOv5 folder. This makes it easier to access data for use as input, as well as to clean up all data (currently in YOLOv5 folder) after it is completed, to avoid wasting usage being provided.



```
[5] import os

# create GWHD folder in yolov5
if not os.path.exists("GWHD"):os.makedirs("GWHD")
!unzip /content/drive/MyDrive/global-wheat/global-wheat-detection.zip -d /content/yolov5/GWHD

inflating: /content/yolov5/GWHD/train/fb6e8bd3f.jpg
inflating: /content/yolov5/GWHD/train/fb804ab78.jpg
inflating: /content/yolov5/GWHD/train/fb9e42d30.jpg
inflating: /content/yolov5/GWHD/train/fbc79aea4.jpg
inflating: /content/yolov5/GWHD/train/fbecfeba8.jpg
inflating: /content/yolov5/GWHD/train/fc0978226.jpg
```

Figure 34. Unzip the dataset to YOLOv5 folder.

The GWHD dataset contains:

- o `train.zip` – 3422 outdoor wheat images use for training.
- o `test.zip` – 10 outdoor wheat images use for test.
- o `train.csv` – training data.

Read the training data in `train.csv` file. Each line represents the data of a bounding box. Due to the high density of wheat, each image may contain multiple wheat heads to be detected, so there may be multiple bounding boxes in an image.

Each column of data respectively corresponds to the unique image id, width, height of image, data of bounding box  $[x_{min}, y_{min}, width, height]$  (Figure 35).

```
[6] import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

df = pd.read_csv('/content/yolov5/GWHD/train.csv')
df.head()
```

	image_id	width	height	bbox	source
0	b6ab77fd7	1024	1024	[834.0, 222.0, 56.0, 36.0]	usask_1
1	b6ab77fd7	1024	1024	[226.0, 548.0, 130.0, 58.0]	usask_1
2	b6ab77fd7	1024	1024	[377.0, 504.0, 74.0, 160.0]	usask_1
3	b6ab77fd7	1024	1024	[834.0, 95.0, 109.0, 107.0]	usask_1
4	b6ab77fd7	1024	1024	[26.0, 144.0, 124.0, 117.0]	usask_1

Figure 35. Reading training data.

The above data file is a common format for bounding boxes in the object detection dataset. The bounding boxes data in YOLO is formatted as  $[class, x_{center}, y_{center}, width, height]$ . The difference can be seen in Figure 36.

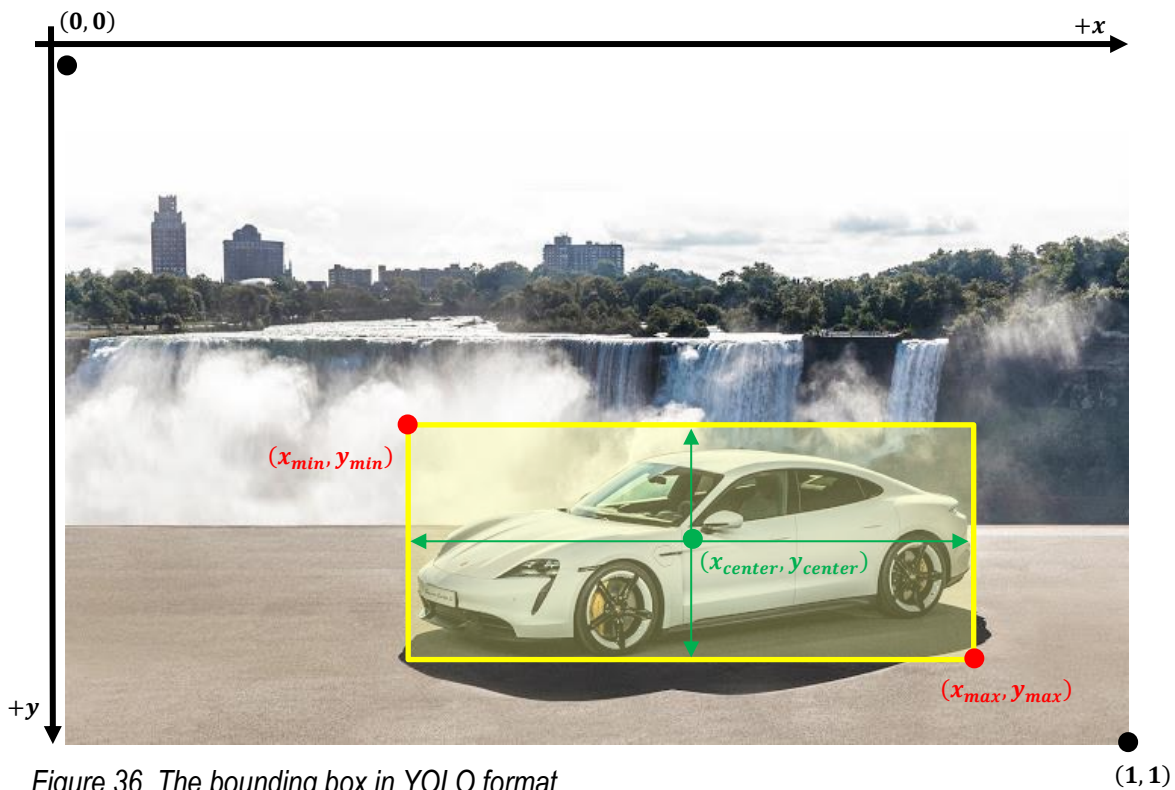


Figure 36. The bounding box in YOLO format.

With minimum point  $(x_{min}, y_{min})$ , the center point  $(x_{center}, y_{center})$  can be calculated as follow:

$$x_{center} = x_{min} + \frac{width}{2}$$
$$y_{center} = y_{center} + \frac{height}{2}$$

Because this is the problem of the single object detection that is the wheat head, the authors of the GWHD dataset do not label the bounding boxes. However, YOLO needs a label parameter, so the bounding boxes need to be labelled corresponding to the classes. Class numbers are zero-indexed (start from 0), therefore label '0' represents the wheat head. All bounding boxes are responsible for detecting the wheat, so they are all labelled '0'.

As shown in *Figure 36*, box coordinates must be normalized in the range 0-1. So,  $x_{center}$  and  $width$  are divided by image width,  $y_{center}$  and  $height$  are divided by image height. Redundant parameters can be removed now.

Now, each column of data respectively corresponds to the unique image id, class,  $x_{center}$ ,  $y_{center}$ , width and height of bounding box as in *Figure 37*.

```

▶ # extract bounding box column into ndarray
bboxes = np.stack(df['bbox'].apply(lambda x: np.fromstring(x[1:-1], sep=',')))
# add new column respectively parameters of bbox
for i, column in enumerate(['xmin', 'ymin', 'w', 'h']):
    df[column] = bboxes[:, i]
# add new column for x,y center point and class bbox belonged to
df['x_center'] = df['xmin'] + df['w']/2
df['y_center'] = df['ymin'] + df['h']/2
df['classes'] = 0
# normalize box coordinates in range 0-1
df['w'] /= 1024
df['h'] /= 1024
df['x_center'] /= 1024
df['y_center'] /= 1024
# only keep these column, remove the remain
df = df[['image_id', 'classes', 'x_center', 'y_center', 'w', 'h']]

```

```
[11] df.head()
```

	image_id	classes	x_center	y_center	w	h
0	b6ab77fd7	0	0.841797	0.234375	0.054688	0.035156
1	b6ab77fd7	0	0.284180	0.563477	0.126953	0.056641
2	b6ab77fd7	0	0.404297	0.570312	0.072266	0.156250
3	b6ab77fd7	0	0.867676	0.145020	0.106445	0.104492
4	b6ab77fd7	0	0.085938	0.197754	0.121094	0.114258

Figure 37. Converting to YOLO format.

### 5.3.1 Creating the label text files

As mentioned previously, each line in data frame `df` represented for a bounding box. That mean, an image can contain multiple boxes.

```

[10] unique_img_ids = df.image_id.unique()
print("Number of unique image id:", len(unique_img_ids))
print(unique_img_ids)

Number of unique image id: 3373
['b6ab77fd7' 'b53afdf5c' '7b72ea0fb' ... 'a5c8d5f5c' 'e6b5e296d'
 '5e0747034']

```

Figure 38. Extract the image id without duplicates.

YOLOv5 in PyTorch reads the bounding box data in file text ( `.txt` ), not file `.csv`. So, the data of all bounding boxes in the same image should be clustered and written in the same file text corresponding to that image. All label text files after writing will be in the `GWHD/label` directory.

```
[12] # create folder for label in txt file
      os.makedirs("/content/yolov5/GWHD/label")

      folder_location = "GWHD/label"
      # loop through all unique image id
      for img_id in unique_img_ids:
          # filter all bbox in df with a specific id
          filt_df = df.query("image_id == @img_id")
          filt_df = filt_df[['classes', 'x_center', 'y_center', 'w', 'h']]

          # create text file in folder_location with name is img_id
          file_name = "{}/{}.txt".format(folder_location, img_id)
          # write all bbox filtered
          with open(file_name, 'a') as file:
              file.write(filt_df.to_string(header = False, index = False))
```

```
[13] # check an example
      !head GWHD/label/b53afdf5c.txt

      0 0.982422 0.809570 0.035156 0.093750
      0 0.357422 0.889160 0.068359 0.092773
      0 0.077637 0.243164 0.063477 0.044922
      0 0.718750 0.975586 0.105469 0.048828
      0 0.137695 0.982422 0.152344 0.035156
      0 0.828613 0.917480 0.047852 0.075195
      0 0.839844 0.438477 0.087891 0.070312
      0 0.541016 0.189453 0.167969 0.064453
      0 0.904297 0.560547 0.074219 0.072266
      0 0.054688 0.119629 0.103516 0.067383
```

Figure 39. Write label text files contain all bounding box data corresponding to each image id.

The GWHD dataset contain 3422 images, but only images containing objects (i.e., bounding boxes) are written in csv files. Meanwhile, images that do not contain or contain near wheat-like objects (such as weeds) can be helpful in training the model, preventing the model from being fooled by similar objects.

Images containing objects are called `positive_images`, and images that do not contain objects are called `negative_images`. In order for the YOLOv5 model to access negative images and use them during training, there also should be label text files corresponding to those negative images. Since these images do not contain any objects, they do not contain any bounding boxes. Therefore, their label text files will be empty.

```
[16] # list of all image names (string)
all_imgs = os.listdir("GWHD/train")
# remove .jpg in image names
all_imgs = [i.split("/")[-1].replace(".jpg", "") for i in all_imgs]
print(all_imgs)

['f91e92fd4', 'f20ba2db5', 'a677f18d9', '701c2e3e9', '3a84eba70', '4bdcb2c78', '3a061fb14',
<
[17] positive_imgs = unique_img_ids # list of images contain bbox
negative_imgs = set(all_imgs) - set(positive_imgs) # list of images not contain bbox
print("Number of image:", len(all_imgs))
print("Number of positive image:", len(positive_imgs))
print("Number of negative image:", len(negative_imgs))

Number of image: 3422
Number of positive image: 3373
Number of negative image: 49
```

Figure 40. Get the list of negative image id.

```
[19] # create txt file in same folder_location with name is negative_img id
for id in list(negative_imgs):
    file_name = "{}/{}.txt".format(folder_location, id)
    with open(file_name, 'w') as f:
        pass
```

Figure 41. Create empty label text files for negative images.

```
[25] path, dirs, files = next(os.walk("GWHD/label"))
file_count = len(files)
print("Number of file in label folder:", file_count)

Number of file in label folder: 3422
```

Figure 42. Final check number of label files. 3422 label text files corresponding to 3422 images.

### 5.3.2 Splitting data into training set and validation set

In order to impartially evaluate the performance of neural network model, a dataset containing the same type of object as the trained object is essential. However, collecting another dataset with a bunch of images and labels is time-consuming, sometimes impossible for an AI researcher. In this case, for example, not all researchers can go to the outdoor wheat field and collect a wheat dataset

themselves to evaluate the model after training is complete. So, they usually use the images directly from the given dataset.

By dividing it proportionally, the common rate is 80-20, with 80% of the dataset used for training and 20% used for evaluation. Thus, model performance will be assessed on 20% of data that was not used during training, or in other words, these data is data which the model has never seen before. This ensures the review will be equitable. The data used in the training process is called the `training set` and the data used in the model evaluation is called the `validation set`.

```
[27] # create folder for training set and validation set
      os.makedirs("GWHd_yolo")
      os.makedirs("GWHd_yolo/images")
      os.makedirs("GWHd_yolo/images/train")
      os.makedirs("GWHd_yolo/images/valid")

      os.makedirs("GWHd_yolo/labels")
      os.makedirs("GWHd_yolo/labels/train")
      os.makedirs("GWHd_yolo/labels/valid")
```

Figure 43. Create training set and validation set folders for images and labels.

```
[28] # shuffle the distribution of data
      np.random.shuffle(all_imgs)
      print("Number of all images:", len(all_imgs))

      # divide train-valid in rate 80-20
      portion = .8
      num_train = round(len(all_imgs) * portion)
      num_valid = len(all_imgs) - num_train
      print("Number of train set, valid set:", num_train, num_valid)

      train_set = all_imgs[:num_train] # list of training image id
      valid_set = all_imgs[num_train:] # list of validation image id
      print(train_set)
      print(valid_set)

      Number of all images: 3422
      Number of train set, valid set: 2738 684
      ['78746472c', '4ae715446', '5a0e2133c', 'cd243b0b0', 'f1016e0ea', '03a988adf', '947117f47',
      ['afd82a1d7', '15c794bfc', 'f1b3454ff', 'b28fb55cc', '0785826af', 'e0a1eadbe', '539ee1a56',
```

Figure 44. Split dataset id into 80% for training and 20% validation set.

Label text files have the same name as the images. That can ensure the labels are moved along with corresponding images to the training set and validation set folders.

```
[31] import shutil

images_sour = "GWHD/train"
labels_sour = "GWHD/label"

images_train_dest = "GWHD_yolo/images/train"
images_valid_dest = "GWHD_yolo/images/valid"
labels_train_dest = "GWHD_yolo/labels/train"
labels_valid_dest = "GWHD_yolo/labels/valid"

# move training images and labels
for name in train_set:
    shutil.move(images_sour + '/' + name + '.jpg', images_train_dest + '/' + name + '.jpg')
    shutil.move(labels_sour + '/' + name + '.txt', labels_train_dest + '/' + name + '.txt')

# move valid images and labels
for name in valid_set:
    shutil.move(images_sour + '/' + name + '.jpg', images_valid_dest + '/' + name + '.jpg')
    shutil.move(labels_sour + '/' + name + '.txt', labels_valid_dest + '/' + name + '.txt')
```

Figure 45. Move images and corresponding labels to training and validation folders.

```
[35] # check file in training folder
img_files = os.listdir("GWHD_yolo/images/train")
print("Number of file in folder:", len(img_files))
print(img_files)

Number of file in folder: 2738
['f20ba2db5.jpg', '701c2e3e9.jpg', '3a84eba70.jpg', '4bdc2c78.jpg', '3a061fb14.jpg',
```

Figure 46. Check name and number of files in these folders.

### 5.3.3 Creating the data.yaml file

The YOLOv5 model on PyTorch accesses the images and uses them as input through a `yaml` file containing summary information about the data set. The `data.yaml` file used in the YOLO model has the following structure:

1. `train:` 'training set directory path'
2. `val:` 'validation set directory path'
- 3.
4. `nc:` 'number of classes'
5. `names:` 'name of objects'



Because the given dataset does not provide a `data.yaml` file, it is necessary to initialize it. Normally, people write this `data.yaml` file in Notepad or Notepad ++, then save it in yaml format and upload to Drive. But it will be written directly in Colab here.

```
[38] import yaml

# create a empty yaml file
with open('data.yaml', 'w') as outfile:
    pass
```

Figure 47. Create empty `data.yaml` file.

To be able to overwrite the empty `yaml` file, there is a function needed to import from `iPython.core.magic`.

```
[39] #customize iPython writefile so we can write variables
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))
```

```
▶ %%writetemplate /content/yolov5/data.yaml
train: /content/yolov5/GWHD_yolo/images/train
val: /content/yolov5/GWHD_yolo/images/valid

nc: 1
names: ['wheat']
```

Figure 48. Overwrite the empty `yaml` file based on the structure.

```
[41] %cat data.yaml

train: /content/yolov5/GWHD_yolo/images/train
val: /content/yolov5/GWHD_yolo/images/valid

nc: 1
names: ['wheat']
```

Figure 49. Check the content of `data.yaml` file.

## 5.4 Training phase

### 5.4.1 Preparing the architecture

Glenn Jocher also provides some sample YOLOv5 models built on previous theory. The YOLOv5 model on PyTorch will read these architectures from the `yaml` file and build it in the `train.py` file. This also makes it easier to configure the architecture depending on the different object detection problems.

```
[43] %cat /content/yolov5/models/yolov5s.yaml

# parameters
nc: 80 # number of classes
depth_multiple: 0.33 # model depth multiple
width_multiple: 0.50 # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23] # P3/8
  - [30,61, 62,45, 59,119] # P4/16
  - [116,90, 156,198, 373,326] # P5/32

# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]], # 0-P1/2
  [-1, 1, Conv, [128, 3, 2]], # 1-P2/4
  [-1, 3, C3, [128]],
  [-1, 1, Conv, [256, 3, 2]], # 3-P3/8
  [-1, 9, C3, [256]],
  [-1, 1, Conv, [512, 3, 2]], # 5-P4/16
  [-1, 9, C3, [512]],
  [-1, 1, Conv, [1024, 3, 2]], # 7-P5/32
  [-1, 1, SPP, [1024, [5, 9, 13]]],
  [-1, 3, C3, [1024, False]], # 9
  ]

# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
  [-1, 1, nn.Upsample, [None, 2, 'nearest']],
  [[-1, 6], 1, Concat, [1]], # cat backbone P4
  [-1, 3, C3, [512, False]], # 13

  [-1, 1, Conv, [256, 1, 1]],
  [-1, 1, nn.Upsample, [None, 2, 'nearest']],
  [[-1, 4], 1, Concat, [1]], # cat backbone P3
  [-1, 3, C3, [256, False]], # 17 (P3/8-small)

  [-1, 1, Conv, [256, 3, 2]],
  [[-1, 14], 1, Concat, [1]], # cat head P4
  [-1, 3, C3, [512, False]], # 20 (P4/16-medium)

  [-1, 1, Conv, [512, 3, 2]],
  [[-1, 10], 1, Concat, [1]], # cat head P5
  [-1, 3, C3, [1024, False]], # 23 (P5/32-large)

  [[17, 20, 23], 1, Detect, [nc, anchors]], # Detect(P3, P4, P5)
  ]
```

Figure 50. Sample YOLOv5 architecture provided by Ultralytics.

The purpose of this thesis is to evaluate the performance of the YOLOv5 algorithm, so the original architecture will be used and will temporarily not configure or add other algorithms and optimization methods to the model.

Because this sample YOLOv5 architecture is used to train the COCO dataset, the number of classes defined is 80. For the wheat dataset, the number of classes needs to be adjusted. Since the anchor box auto-learning has been integrated, the anchor box parameters can be ignored as default.

```
[45] num_classes = 1

[46] %%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes} # number of classes
depth_multiple: 0.33 # model depth multiple
width_multiple: 0.50 # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23] # P3/8
  - [30,61, 62,45, 59,119] # P4/16
  - [116,90, 156,198, 373,326] # P5/32
```

Figure 51. Overwrite the number of classes and save as custom model.

## 5.4.2 Training model

With the command line shown in Figure 52, the model will be trained by compile file `train.py` along with its configurable arguments.

```
[48] # train yolov5s on custom data for 100 epochs
# time its performance
%%time
%cd /content/yolov5/
!python train.py --img 416 --batch 32 --epochs 100 --data data.yaml \
    --cfg ./models/custom_yolov5s.yaml --weights '' --name wheat5s_results --cache
```

Figure 52. Implement the training process.

These following arguments represent for:

- **img**: define input image size. The original image size is  $1024 \times 1024$ , compress to smaller size make the training process faster. After many experiments, many computer

vision researchers agreed that the size  $416 \times 416$  is the ideal size to use as input without losing much detail.

- **batch:** determine the batch size. The forwarding of thousand images into the neural network at the same time makes the number of weights that the model learns in one time (one epoch) to increase a lot. Thus, the dataset is usually divided into multiple batches of  $n$  images and training batch by batch. The results of each batch are then saved to RAM and aggregated after the training for all batches is completed. Because the weights learned from the batches are stored in RAM, so the larger the number of batches, the more memory consumption will be consumed.

The training set contains 2738 images, with *batch size* = 32, the number of batches will be  $2738 \div 32 = 86$  *batches*.

- **epochs:** define the number of training epochs. An epoch is responsible for learning all input images, in other words, training all input. Since the dataset is split into multiple batches, one epoch will be responsible for training all the batches. The number of epochs represents the number of times the model trains all the inputs and updates the weights to get closer to the ground truth labels. Often chosen based on experience and intuition. The number of epochs more than 3000 is normal.
- **data:** the path to `data.yaml` file containing the summary of the dataset. The model evaluation process is executed immediately after each epoch, so the model will also access the validation directory via the path in `data.yaml` file and use its contents for evaluation at that moment.
- **cfg:** specify our model configuration path. Based on the architecture defined in the model `yaml` file previously, this command line allows the `train.py` file to compile and build this architecture for training input images.
- **weights:** specify a path to weights. A pretrained weight can be used for saving training time. If it is left blank, the model will automatically initialize random weights for training.

- **name:** name of result folder. The model will create a directory containing all the results performed during training.
- **cache:** cache images for training faster.

```

Epoch 90/99  gpu_mem 2.54G  box 0.04644  obj 0.1807  cls 0  total 0.2272  targets 1216  img_size 416: 100% 86/86 [00:10<00:00, 8.14it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.54it/s]
              all 684 2.95e+04 0.652 0.941 0.935 0.505

Epoch 91/99  gpu_mem 2.54G  box 0.04635  obj 0.1807  cls 0  total 0.2271  targets 1028  img_size 416: 100% 86/86 [00:10<00:00, 8.05it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.60it/s]
              all 684 2.95e+04 0.672 0.939 0.935 0.508

Epoch 92/99  gpu_mem 2.54G  box 0.04622  obj 0.1788  cls 0  total 0.2251  targets 1024  img_size 416: 100% 86/86 [00:10<00:00, 8.23it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.62it/s]
              all 684 2.95e+04 0.667 0.939 0.935 0.505

Epoch 93/99  gpu_mem 2.54G  box 0.04629  obj 0.1801  cls 0  total 0.2264  targets 963  img_size 416: 100% 86/86 [00:10<00:00, 8.32it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:09<00:00, 2.39it/s]
              all 684 2.95e+04 0.658 0.94 0.935 0.502

Epoch 94/99  gpu_mem 2.54G  box 0.04637  obj 0.1802  cls 0  total 0.2266  targets 796  img_size 416: 100% 86/86 [00:10<00:00, 8.15it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.63it/s]
              all 684 2.95e+04 0.664 0.94 0.935 0.504

Epoch 95/99  gpu_mem 2.54G  box 0.04641  obj 0.1823  cls 0  total 0.2287  targets 1165  img_size 416: 100% 86/86 [00:10<00:00, 8.17it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:09<00:00, 2.43it/s]
              all 684 2.95e+04 0.646 0.941 0.935 0.503

Epoch 96/99  gpu_mem 2.54G  box 0.04602  obj 0.1763  cls 0  total 0.2223  targets 1032  img_size 416: 100% 86/86 [00:10<00:00, 8.14it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:09<00:00, 2.39it/s]
              all 684 2.95e+04 0.658 0.941 0.936 0.508

Epoch 97/99  gpu_mem 2.54G  box 0.04607  obj 0.1779  cls 0  total 0.224  targets 1132  img_size 416: 100% 86/86 [00:10<00:00, 8.21it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.62it/s]
              all 684 2.95e+04 0.651 0.94 0.935 0.503

Epoch 98/99  gpu_mem 2.54G  box 0.04619  obj 0.18  cls 0  total 0.2262  targets 1053  img_size 416: 100% 86/86 [00:10<00:00, 8.05it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:08<00:00, 2.56it/s]
              all 684 2.95e+04 0.659 0.941 0.935 0.505

Epoch 99/99  gpu_mem 2.54G  box 0.04604  obj 0.1788  cls 0  total 0.2248  targets 777  img_size 416: 100% 86/86 [00:10<00:00, 8.12it/s]
              Class Images Targets  P  R  mAP@.5  mAP@.5:.95: 100% 22/22 [00:10<00:00, 2.03it/s]
              all 684 2.95e+04 0.663 0.94 0.935 0.505

Optimizer stripped from runs/train/wheat5s_results/weights/last.pt, 14.7MB
Optimizer stripped from runs/train/wheat5s_results/weights/best.pt, 14.7MB
100 epochs completed in 0.581 hours.

CPU times: user 12.3 s, sys: 3.82 s, total: 16.1 s
Wall time: 35min 41s

```

Figure 53. Training progress in 100 epochs.

For 1 epoch, the average time to perform the training process on 86 batches were 10 seconds and for evaluation on 22 batches were 9 seconds. The total execution time was 35 minutes and 41 seconds for 100 epochs on the dataset containing 3422 images and the mAP of last epoch is 93.5%.

The weighting result obtained in the last epoch is not always the weight for the highest accuracy. Thus, an add-in called TensorBoard was created that clearly visualized the entire training process.

```
[38] # Start tensorboard
      # Launch after you have started training
      # logs save in the folder "runs"
      %load_ext tensorboard
      %tensorboard --logdir runs
```

Figure 54. Use TensorBoard to load the entire training process saved in runs folder.

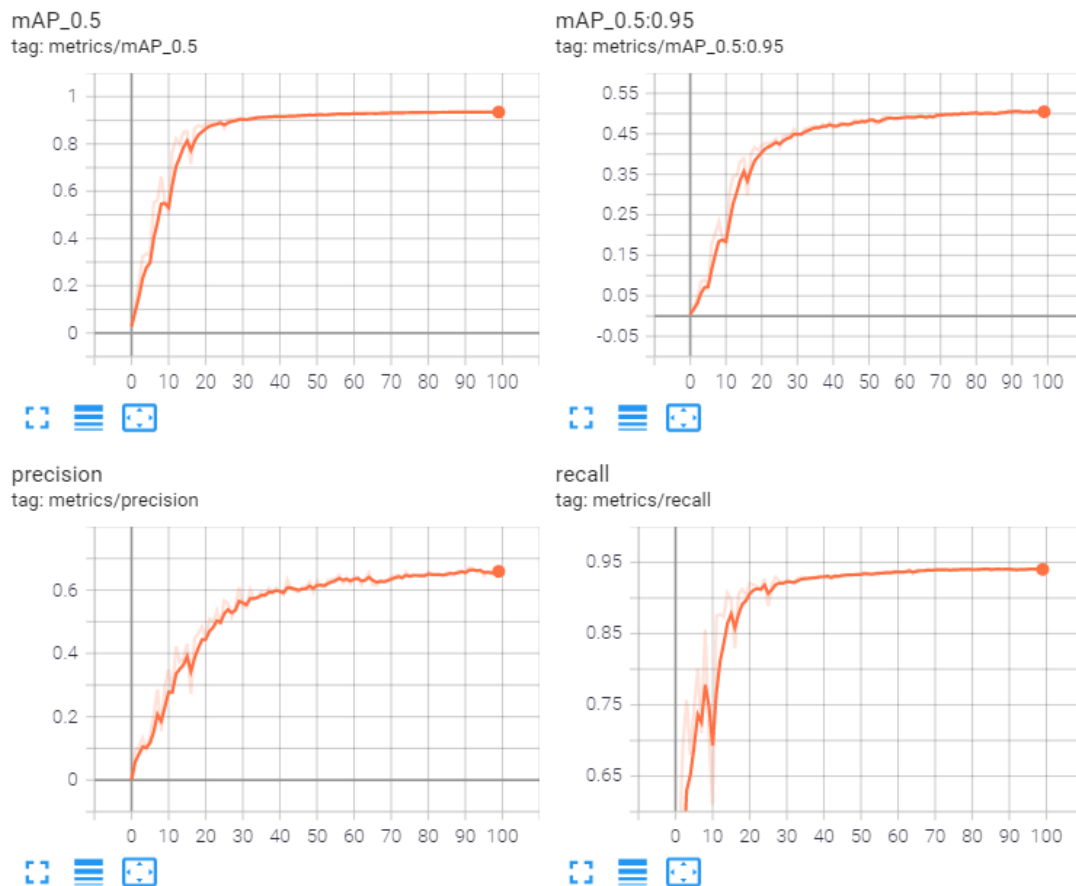


Figure 55. The evaluation metrics of the entire 100 epochs were visualized in graphs.

As shown in Figure 53 and Figure 55, with a dataset containing 3422 images, the model takes about 20 seconds to complete one epoch, and only with 100 epochs, the accuracy of model is about 93%. This proves that, with the mere original architecture of YOLOv5, the model is not only fast, but the accuracy is also high without any optimization methods integrated into it.

Besides, the model saves 2 weighting results as `pt` file. As shown in Figure 56, `last.pt` file is the weight at the last epoch and `best.pt` file is the weight at the last epoch for the highest accuracy. The size of both files is only 14MB, making it very light to integrate into AI systems (in web or mobile application) as a pretrained weight while maintaining 93.5% accuracy.

```
[41] %ls runs/train/wheat5s_results/weights
```

```
best.pt last.pt
```

Figure 56. Two result weight files were exported for use later.

## 5.5 Inference with trained weight

Trained weights can be used to identify the wheat head on any image. If the presence of a wheat head is detected, a bounding box is drawn to encase the object and display the probability that the object is the head of wheat.

The GWHD dataset provides 10 outdoor wheat images which non-duplicated with 3422 images that were used in the training. These images are also not labelled the ground truth bounding boxes. They can be observed as images of a farmer taken in their field and tested with weights trained previously in the wheat head detection model.

The method of performing object detection with trained weights is similar to training the model. Using the command shown in Figure 57, `detect.py` file will be compiled, and it rebuilds the architecture used in the training. Trained weights will be used to predict objects and limit boxes for them with 93.5% accuracy.

```
[47] # use the best weights!
%cd /content/yolov5/
!python detect.py --weights runs/train/wheat5s_results/weights/best.pt --img 416 \
    --conf 0.4 --source GWHD/test

/content/yolov5
Namespace(agnostic_nms=False, augment=False, classes=None, conf_thres=0.4, device='', exist_ok=False,
YOLOv5 v4.0-12-g509dd51 torch 1.7.0+cu101 CUDA:0 (Tesla V100-SXM2-16GB, 16130.5MB)

Fusing layers...
Model Summary: 232 layers, 7246518 parameters, 0 gradients, 16.8 GFLOPS
image 1/10 /content/yolov5/GWHD/test/2fd875eaa.jpg: 416x416 26 wheats, Done. (0.013s)
image 2/10 /content/yolov5/GWHD/test/348a992bb.jpg: 416x416 37 wheats, Done. (0.014s)
image 3/10 /content/yolov5/GWHD/test/51b3e36ab.jpg: 416x416 25 wheats, Done. (0.015s)
image 4/10 /content/yolov5/GWHD/test/51f1be19e.jpg: 416x416 18 wheats, Done. (0.011s)
image 5/10 /content/yolov5/GWHD/test/53f253011.jpg: 416x416 29 wheats, Done. (0.011s)
image 6/10 /content/yolov5/GWHD/test/796707dd7.jpg: 416x416 14 wheats, Done. (0.011s)
image 7/10 /content/yolov5/GWHD/test/aac893a91.jpg: 416x416 20 wheats, Done. (0.011s)
image 8/10 /content/yolov5/GWHD/test/cb8d261a3.jpg: 416x416 22 wheats, Done. (0.012s)
image 9/10 /content/yolov5/GWHD/test/cc3532ff6.jpg: 416x416 26 wheats, Done. (0.012s)
image 10/10 /content/yolov5/GWHD/test/f5a1f0358.jpg: 416x416 25 wheats, Done. (0.013s)
Results saved to runs/detect/exp3
Done. (0.588s)
```

Figure 57. Detect wheat head with trained weight.

After the detection is complete, the predicted bounding boxes that cover the objects (wheat heads) will be drawn into the image. They will be saved in the same folder containing results from the training phase.

```
[63] import glob
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# load image in ndarray format
images = []
for img_path in glob.glob('/content/yolov5/runs/detect/exp5/*.jpg'):
    images.append(mpimg.imread(img_path))
```

Figure 58. Load predicted images as array format for visualization.

```
[64] %matplotlib inline

# create subplot
fig, axs = plt.subplots(2, 2, figsize=(20, 20), sharex='col', sharey='row',
                        gridspec_kw={'hspace': .05, 'wspace': 0.05})

axs = axs.flatten()
for img, ax in zip(images, axs):
    ax.imshow(img)
plt.show()
# save figure as .png file
fig.savefig('/content/yolov5/runs/fig.png', bbox_inches='tight')
```

Figure 59. Visualize the predicted images by trained weight.



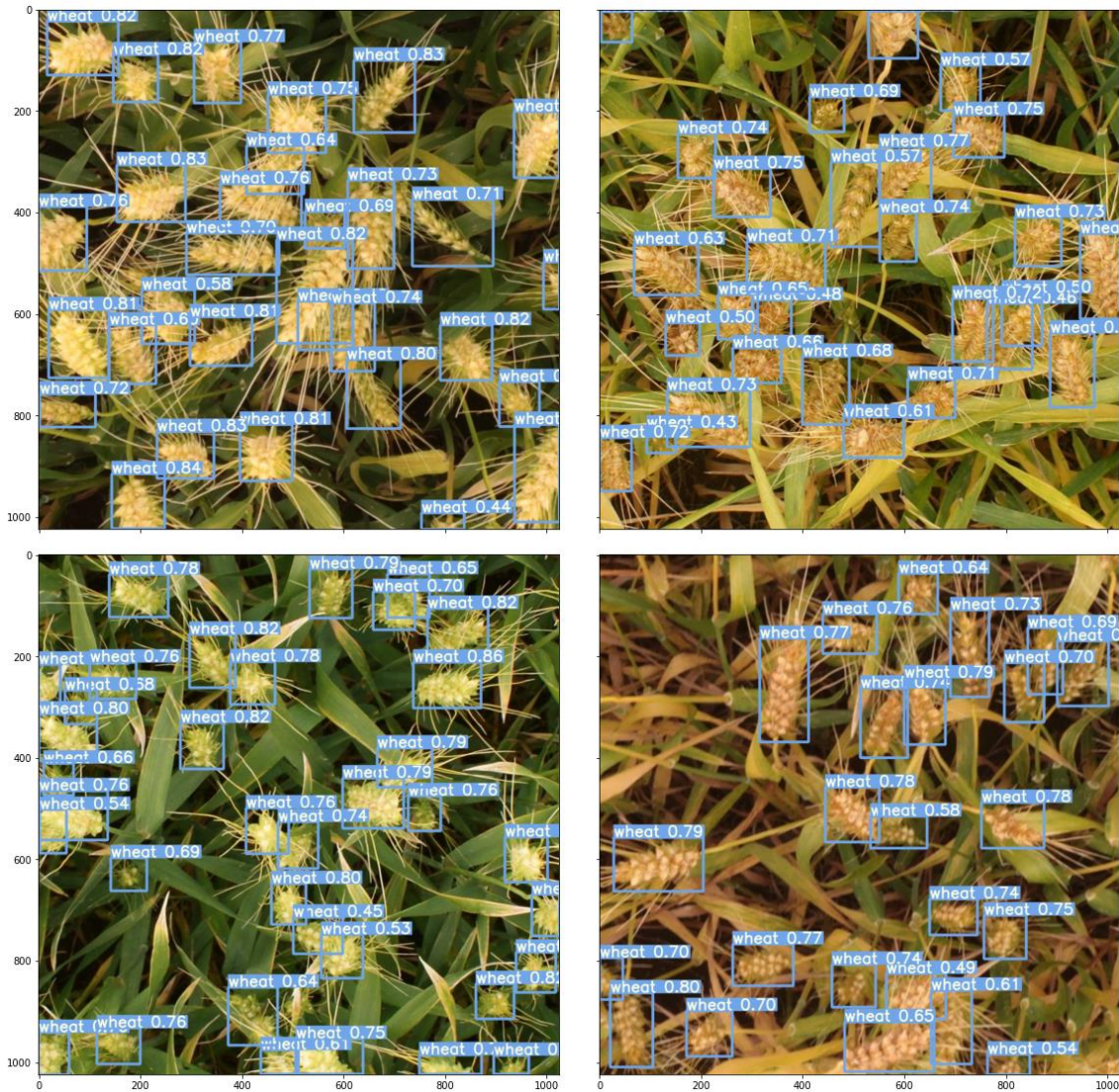


Figure 60. Four of ten predicted images were detected by using best trained weight.

As shown in Figure 60, the model gives out extremely impressive predictive results on even the images it has not seen before. Although the density of wheat heads in each image is very high, the model almost detects that all wheat heads appear in the image. However, because the model's accuracy was only 93.5%, some wheat heads were still missing in detection. As can be seen in Figure 60, although the wheat heads are correctly predicted, the probability for them is not high.

As mentioned, the model predicts an object based on the probability for that object, if the probability is less than a given threshold (here is 0.4) then the model predicts it is not a wheat head. With a low prediction probability near the threshold, some wheat heads may have a probability lower than the threshold and be predicted not a wheat head. Typically, in one of the 10 predicted images

(Figure 61), some of the wheat in the bottom left and center of the image is not predicted to be wheat even their presences are clear.

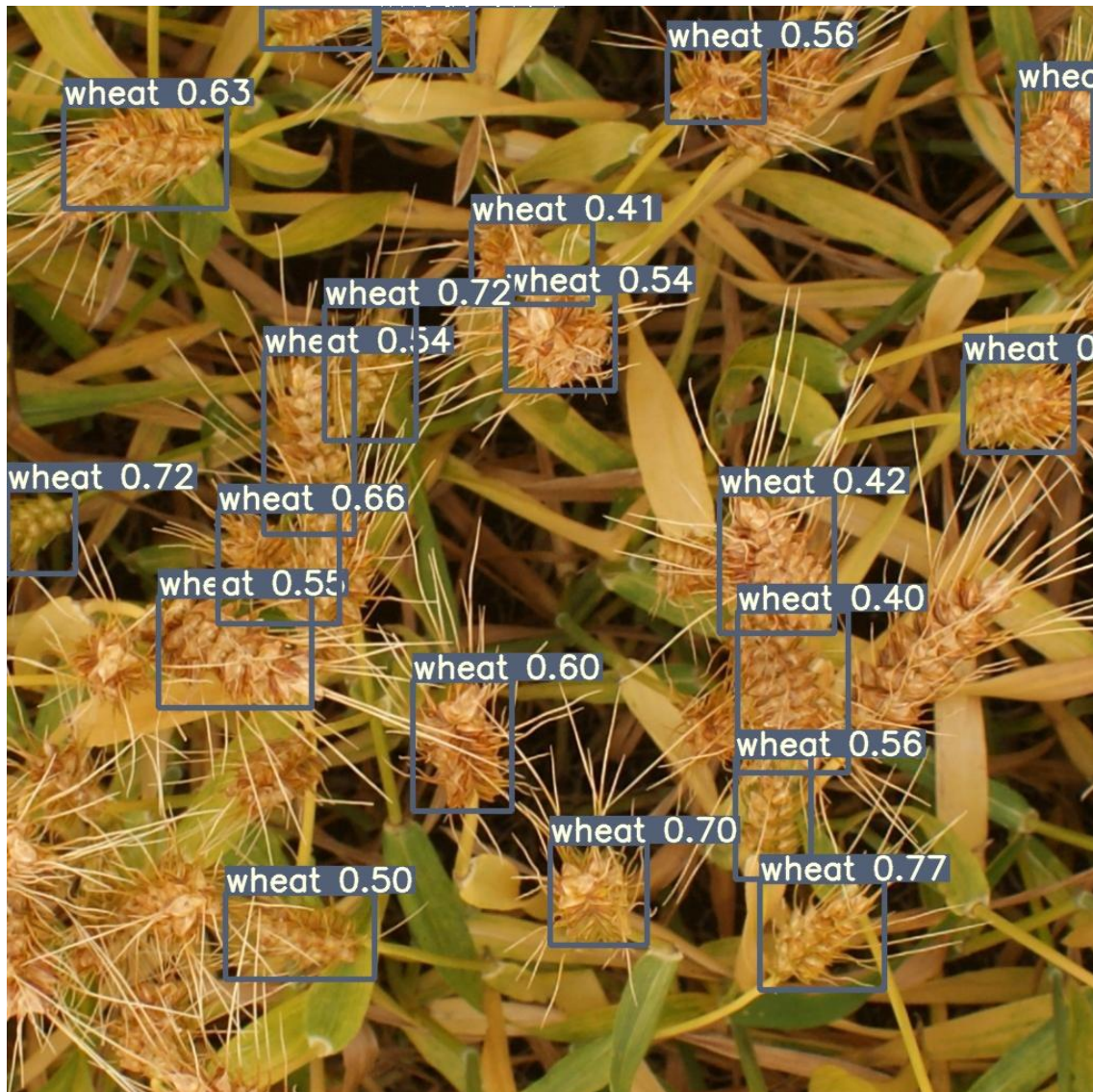


Figure 61. One of ten predicted image which has missing detection.

## 6 CONCLUSION

Nowadays, there is still a lot of controversy about the name and improvements of YOLOv5 in the computer vision community about innovations that have not really made a breakthrough. However, the name aside, the performance of YOLOv5 is at least not inferior to the YOLOv4 in both speed and accuracy. With the built-in Pytorch framework that is user-friendly and has a larger community than the Darknet framework, there is no doubt that YOLOv5 will receive more contributions and have more growth potential in the future.

Actually, the field of computer vision, especially object detection, has only exploded in the last 5 years or so. Therefore, although it has evolved over 5 generations and is one of the outstanding object detection algorithms, the YOLO algorithm is still flawed. Therefore, an AI system cannot be built from a mere algorithm, it is necessary to integrate more optimization methods and the most state-of-the-art ideas in the field of computer vision to help the AI system achieve the best performance.

## 7 REFERENCES

- Bochkovskiy, A.** (2020, April). Yolo v4, v3 and v2 for Windows and Linux. (GitHub) Seach date 27.11.2020. <https://github.com/AlexeyAB/darknet>
- Bochkovskiy, A., Wang, C.-Y., & Mark Liao, H.-Y.** (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv. Seach date 24.11.2020. <https://arxiv.org/pdf/2004.10934.pdf>
- Datahacker.rs.** (2018, 11). Data hacker. Seach date 12.11.2020. <http://datahacker.rs/deep-learning-bounding-boxes/>
- E. David, S. M.-T.** (2020). Global Wheat Head Detection (GWHD) dataset: a large and diverse dataset of high resolution RGB labelled images to develop and benchmark wheat head detection methods. arXiv. Seach date 07.12.2020. <https://arxiv.org/ftp/arxiv/papers/2005/2005.02162.pdf>
- Gochoo, M.** (2020). ReseachGate. Search date 03.12.2020. researchgate.net: [https://www.researchgate.net/figure/a-Feature-pyramid-network-FPN-b-YOLO3-c-Proposed-concatenated-feature-pyramid\\_fig2\\_335538302](https://www.researchgate.net/figure/a-Feature-pyramid-network-FPN-b-YOLO3-c-Proposed-concatenated-feature-pyramid_fig2_335538302)
- He, K., Zhang, X., Ren, S., & Sun, J.** (2015). Deep Residual Learning for Image Recognition. arXiv. Seach date 22.11.2020. <https://arxiv.org/pdf/1512.03385.pdf>
- He, K., Zhang, X., Ren, S., & Sun, J.** (2015). Spatial Pyramid Pooling in Deep Convolutional. arXiv. Seach date 01.12.2020. <https://arxiv.org/pdf/1406.4729.pdf>
- Huang, G., Liu, Z., & Maaten, L. v.** (2018). Densely Connected Convolutional Networks. arXiv. Seach date 28.11.2020. <https://arxiv.org/pdf/1608.06993.pdf>
- Huang, Z., & Wang, J.** (2019). DC-SPP-YOLO: Dense Connection and Spatial Pyramid Pooling Based YOLO for Object Detection. arXiv. Seach date 30.11.2020. <https://arxiv.org/ftp/arxiv/papers/1903/1903.08589.pdf>
- Hui, J.** (2020). YOLOv4. Medium. Seach date 27.11.2020. <https://jonathan-hui.medium.com/yolov4-c9901eaa8e61>
- Ioffe, S., & Szegedy, C.** (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv. Seach date 20.11.2020. <https://arxiv.org/pdf/1502.03167.pdf>
- Jocher, G.** (2020, June). YOLOv5. (GitHub) Seach date 10.11.2020. doi: [https://zenodo.org/record/4418161#.X\\_iH\\_ugzaUk](https://zenodo.org/record/4418161#.X_iH_ugzaUk)
- Kamal, A.** (2019). YOLO, YOLOv2 and YOLOv3: All You want to know. Medium. Seach date 18.11.2020. [https://medium.com/@amrokamal\\_47691/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899](https://medium.com/@amrokamal_47691/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899)

- Kanjee, R.** (2020). YOLOv4 — Superior, Faster & More Accurate Object Detection. Medium. Search date 06.12.2020. <https://augmentedstartups.medium.com/yolov4-superior-faster-more-accurate-object-detection-7e8194bf1872>
- Kathuria, A.** (2018). What's new in YOLO v3? Medium. Search date 23.11.2020. <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>
- Liu, S., Qi, L., Qin, H., Shi, J., & Jia, J.** (2018). Path Aggregation Network for Instance Segmentation. arXiv. Search date 03.12.2020. <https://arxiv.org/pdf/1803.01534.pdf>
- Menegaz, M.** (2018). Understanding YOLO. hackernoon. Search date 12.11.2021. <https://hackernoon.com/understanding-yolo-f5a74bbc7967>
- ODSC Science.** (2018). Overview of the YOLO Object Detection Algorithm. Medium. Search date 12.11.2020. <https://medium.com/@ODSC/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0>
- Redmon, J., & Farhadi, A.** (2016). YOLO9000: Better, Faster, Stronger. arXiv. Search date 17.11.2020. <https://arxiv.org/pdf/1612.08242.pdf>
- Redmon, J., & Farhadi, A.** (2018). YOLOv3: An Incremental Improvement. arXiv. Search date 20.11.2020. <https://arxiv.org/pdf/1804.02767.pdf>
- Redmon, J., Divvala, S., Girshick, R., & Farhadi, A.** (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv. Search date 12.11.2020. <https://arxiv.org/pdf/1506.02640.pdf>
- Shah, D.** (2020). YOLOv4 — Version 3: Proposed Workflow. Search date 02.12.2020. <https://medium.com/visionwizard/yolov4-version-3-proposed-workflow-e4fa175b902>
- Solawetz, J.** (2020). Breaking Down YOLOv4. Roboflow. Search date 27.11.2020. <https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/>
- Solawetz, J.** (2020). YOLOv5 New Version - Improvements And Evaluation. Roboflow. Search date 04.2020. <https://blog.roboflow.com/yolov5-improvements-and-evaluation/>
- V Thatte, A.** (2020). Evolution of YOLO — YOLO version 1. Medium. Search date 14.11.2020. <https://towardsdatascience.com/evolution-of-yolo-yolo-version-1-afb8af302bd2>
- Wang, C.-Y., Mark Liao, H.-Y., Yeh, I.-H., Wu, Y.-H., Chen, P.-Y., & Hsieh, J.-W.** (2019). CSP-NET: A new backbone that can enhance learning capability of CNN. arXiv. Search date 30.11.2020. <https://arxiv.org/pdf/1911.11929.pdf>
- Yanjia LiYanjia Li, E.** (2019). Dive Really Deep into YOLO v3: A Beginner's Guide. Medium. Search date 22.11.2020. <https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e>