Project By: Trapti Khandelwal | Mohnish Lavania | Sri Lalana | Rahul Singh

REAL TIME OBJECT DETECTION (BASIC-I)

GUIDED BY: VEDANT DWIVEDI | SOMESH CHOUDHARY

Report

Introduction

Making a general object detection model(pretrained) such that it can be used in flutter applications and other web application calling the model via api(restful).

Object detection models seek to identify the presence of relevant objects in images and classify those objects into relevant classes.

- Basic Approach:- Our basic approach was converting a pre-trained yolov5 object detection model .

What is YOLO Object Detection?

YOLO ("You Only Look Once") is an effective real-time object recognition algorithm, first described in the seminal 2015 paper by Joseph Redmon et al. In this article we introduce the concept of object detection, the YOLO algorithm itself, and one of the algorithm's open source implementations: Darknet.
Image classification is one of the many exciting applications of convolutional neural networks. Aside from simple image classification, there are plenty of fascinating problems in computer vision, with object detection being one of the most interesting. It is commonly associated with self-driving cars where systems blend computer vision, LIDAR and other technologies to generate a multidimensional representation of the road with all its participants. Object detection is also commonly used in video surveillance, especially in crowd monitoring to prevent terrorist attacks, count people for general statistics or analyze customer experience with walking paths within shopping centers.

Object Detection Overview

To explore the concept of object detection it is useful to begin with image classification. It goes through levels of incremental complexity.
Image classification (1) aims at assigning an image to one of a number of different categories (e.g. car, dog, cat, human, etc.), essentially answering the question "What is in this picture?". One image has only one category assigned to it.
Object localization (2) then allows us to locate our object in the image, so our question changes to "What is it and where it is?".
In a real real-life scenario, we need to go beyond locating just one object but rather multiple objects in one image. For example, a self-driving car has to find the location of other cars, traffic lights, signs, humans and to take appropriate action based on this information.

YOLO algorithm

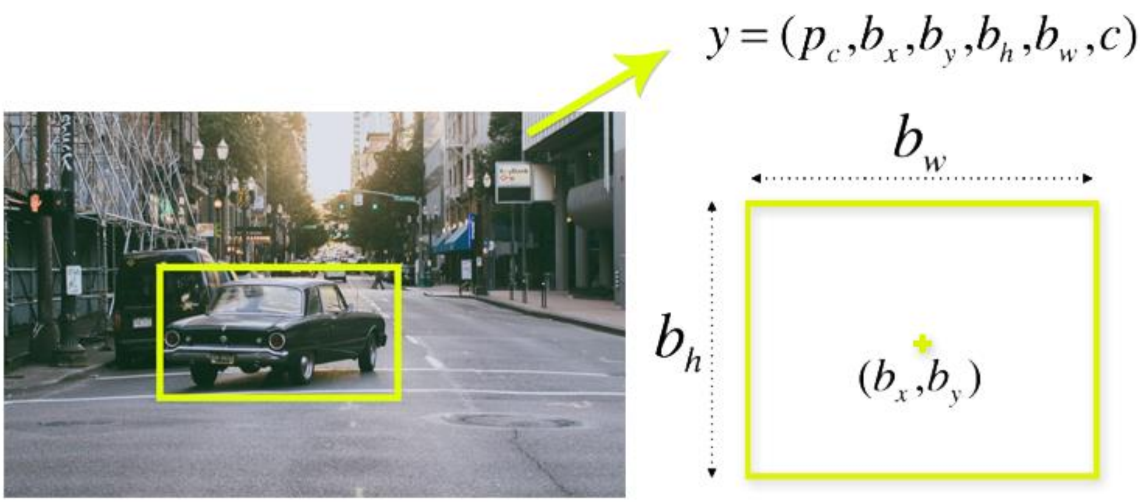
There are a few different algorithms for object detection and they can be split into two groups:

- Algorithms based on classification. They are implemented in two stages. First, they select regions of interest in an image. Second, they classify these regions using convolutional neural networks. This solution can be slow because we have to run predictions for every selected region. A widely known example of this type of algorithm is the Region-based convolutional neural network (RCNN) and its cousins Fast-RCNN, Faster-RCNN and the latest addition to the family: Mask-RCNN. Another example is RetinaNet.
- Algorithms based on regression – instead of selecting interesting parts of an image, they predict classes and bounding boxes for the whole image in one run of the algorithm. The two best known examples from this group are the YOLO (You Only Look Once) family algorithms and SSD (Single Shot Multibox Detector). They are commonly used for real-time object detection as, in general, they trade a bit of accuracy for large improvements in speed.

To understand the YOLO algorithm, it is necessary to establish what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

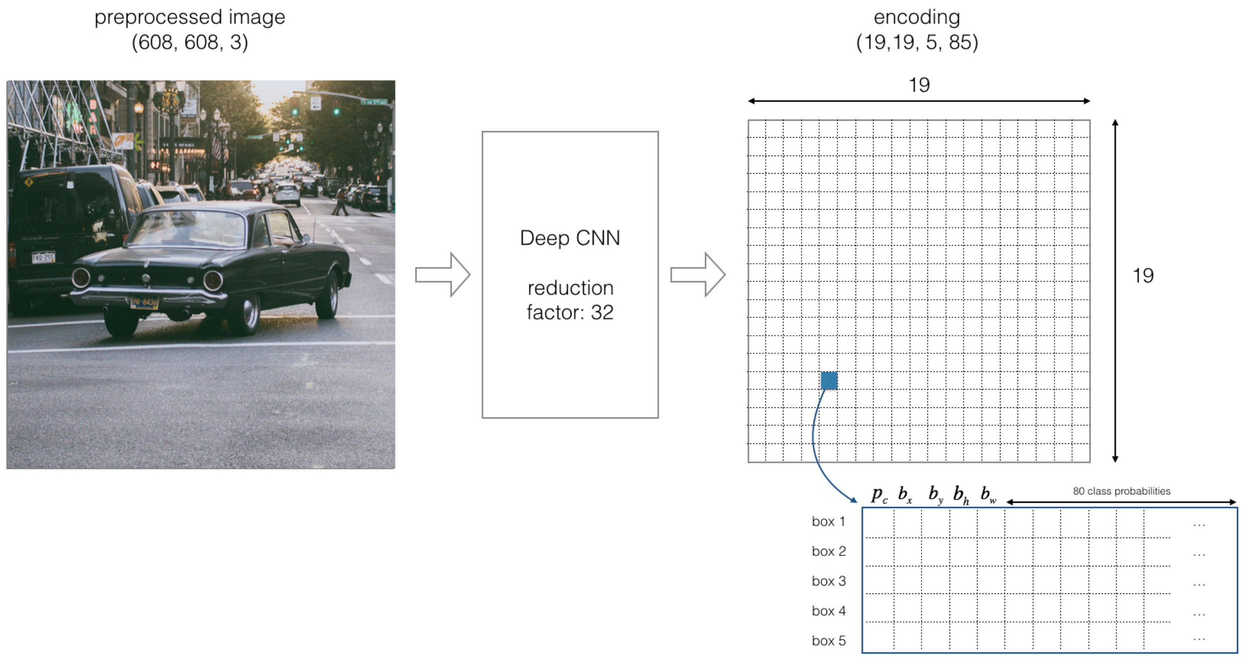
- center of a bounding box (bxby)
- width (bw)
- height (bh)
- value cis corresponding to a class of an object (such as: car, traffic lights, etc.).

In addition, we have to predict the pc value, which is the probability that there is an object in the boundingbox.



As we mentioned above, when working with the YOLO algorithm we are not searching for interesting regions in our image that could potentially contain an object.

Instead, we are splitting our image into cells, typically using a 19×19 grid. Each cell is responsible for predicting 5 bounding boxes (in case there is more than one object in this cell). Therefore, we arrive at a large number of 1805 bounding boxes for one image.



Most of these cells and bounding boxes will not contain an object. Therefore, we predict the value pc, which serves to remove boxes with low object probability and bounding boxes with the highest shared area in a process called non-max suppression.

Darknet – a YOLO implementation

There are a few different implementations of the YOLO algorithm on the web. Darknet is one such open source neural network framework (a PyTorch implementation can be found here or with some extra fast.ai functionality here; a Keras implementation can be found here). Darknet was written in the C Language and CUDAtechnology, which makes it really fast and provides for making computations on a GPU, which is essential for real-time predictions.

Installation is simple and requires running just 3 lines of code (in order to use GPU it is necessary to modify the settings in the Makefile script after cloning the repository).

After installation, we can use a pre-trained model or build a new one from scratch. For example here's how you can detect objects on your image using model pre-trained on COCO dataset:

Introduction to YOLO v5

YOLOv5 is a recent release of the YOLO family of models. YOLO v5 is the first of YOLO models to be written in the PyTorch framework and it is much more lightweight and easy to use. It is much easier to get started with and offers you much greater development speed when moving into deployment.

Installing the YOLOv5 Environment

To start off with YOLOv5 we first clone the YOLOv5 repository and install dependencies. This will set up our programming environment to be ready to running object detection training and inference commands.

```
!git clone https://github.com/ultralytics/yolov5  # clone repo
!pip install -U -r yolov5/requirements.txt  # install dependencies%cd /content/yolov5
```

Then, we can take a look at our training environment provided to us for free from Google Colab.

```
import torch
from IPython.display import Image  # for displaying images
from utils.google_utils import gdrive_download  # for downloading models/datasetsprint('torch %s %s' % (torch.__version__,
torch.cuda.get_device_properties(0) if torch.cuda.is_available() else 'CPU'))
```

It is likely that you will receive a Tesla P100 GPU from Google Colab. Here is what We received:

```
torch 1.5.0+cu101 _CudaDeviceProperties(name='Tesla P100-PCIE-16GB', major=6, minor=0, total_memory=16280MB,
multi_processor_count=56)
```

The GPU will allow us to accelerate training time. Colab is also nice in that it come preinstalled with torch and cuda.

Download Custom YOLOv5 Object Detection Data

In this tutorial we will download custom object detection data in YOLOv5 format from Roboflow. You can follow along with the public blood cell dataset or upload your own dataset.
Once you have labeled data, to get move your data into Roboflow, create a free account and then you can drag your dataset in in any format: (VOC XML, COCO JSON, TensorFlow Object Detection CSV, etc).

Once uploaded you can choose preprocessing and augmentation steps:

The settings chosen for the BCCD example dataset
Then, click Generate and Download and you will be able to choose YOLOv5 PyTorch format.
Select "YOLO v5 PyTorch"
When prompted, be sure to select "Show Code Snippet." This will output a download curl script so you can easily port your data into Colab in the proper format.

```
curl -L "https://public.roboflow.ai/ds/YOUR-LINK-HERE" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

Downloading in Colab…

Download a custom object detection dataset in YOLOv5 format
The export creates a YOLOv5 .yaml file called data.yaml specifying the location of a YOLOv5 images folder, a YOLOv5 labels folder, and information on our custom classes.

Define YOLOv5 Model Configuration and Architecture

Next we write a model configuration file for our custom object detector. For this tutorial, we chose the smallest, fastest base model of YOLOv5. You have the option to pick from other YOLOv5 models including:

- YOLOv5s
- YOLOv5m
- YOLOv5l
- YOLOv5x

You can also edit the structure of the network in this step, though rarely will you need to do this. Here is the YOLOv5 model configuration file, which we term custom_yolov5s.yaml:

```
nc: 3
depth_multiple: 0.33
width_multiple: 0.50anchors:
  - [10,13, 16,30, 33,23]
  - [30,61, 62,45, 59,119]
  - [116,90, 156,198, 373,326]backbone:
  [[-1, 1, Focus, [64, 3]],
   [-1, 1, Conv, [128, 3, 2]],
   [-1, 3, Bottleneck, [128]],
   [-1, 1, Conv, [256, 3, 2]],
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 6, BottleneckCSP, [1024]],
  ]head:
  [[-1, 3, BottleneckCSP, [1024, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],
   [-2, 1, nn.Upsample, [None, 2, "nearest"]],
   [[-1, 6], 1, Concat, [1]],
   [-1, 1, Conv, [512, 1, 1]],
   [-1, 3, BottleneckCSP, [512, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],
   [-2, 1, nn.Upsample, [None, 2, "nearest"]],
   [[-1, 4], 1, Concat, [1]],
   [-1, 1, Conv, [256, 1, 1]],
   [-1, 3, BottleneckCSP, [256, False]],
   [-1, 1, nn.Conv2d, [na * (nc + 5), 1, 1, 0]],[[], 1, Detect, [nc, anchors]],
  ]
```

Training Custom YOLOv5 Detector

With our data.yaml and custom_yolov5s.yaml files ready to go we are ready to train!
To kick off training we running the training command with the following options:

- img: define input image size
- batch: determine batch size
- epochs: define the number of training epochs. (Note: often, 3000+ are common here!)
- data: set the path to our yaml file
- cfg: specify our model configuration
- weights: specify a custom path to weights. (Note: you can download weights from the Ultralytics Google Drive folder)
- name: result names
- nosave: only save the final checkpoint
- cache: cache images for faster training

And run the training command:

Training a custom YOLOv5 detector. It trains quickly!

During training, you want to be watching the mAP@0.5 to see how your detector is learning to detect on your validation set, higher is better!

Evaluate Custom YOLOv5 Detector Performance

Now that we have completed training, we can evaluate how well the training procedure performed by looking at the validation metrics. The training script will drop tensorboard logs in runs. We visualize those here:
Visualizing tensorboard results on our custom dataset. And if you can't visualize Tensorboard for whatever reason the results can also be plotted with utils.plot_results and saving a result.png. Training plots in .png format

Run YOLOv5 Inference on Test Images

Now we take our trained model and make inference on test images. After training has completed model weights will save in weights/. For inference we invoke those weights along with a conf specifying model confidence (higher confidence required makes less predictions), and a inference source. source can accept a directory of images, individual images, video files, and also a device's webcam port. For source, I have moved our test/*jpg to test_infer/.

```
!python detect.py --weights weights/last_yolov5s_custom.pt --img 416 --conf 0.4 --source ../test_infer
```

The inference time is extremely fast. On our Tesla P100, the YOLOv5s is hitting 7ms per image. This bodes well for deploying to a smaller GPU like a Jetson Nano (which costs only $100).
Inference on YOLOv5s occurring at 142 FPS (.007s/image)

Finally, we visualize our detectors inferences on test images.

YOLOv5 inference on test images. It can also easily infer on video and webcam.

Recall and Precision

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.352
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.544
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.378
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.187
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.397
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.459
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.296
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.496
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.557
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.358
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.619
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.700
```

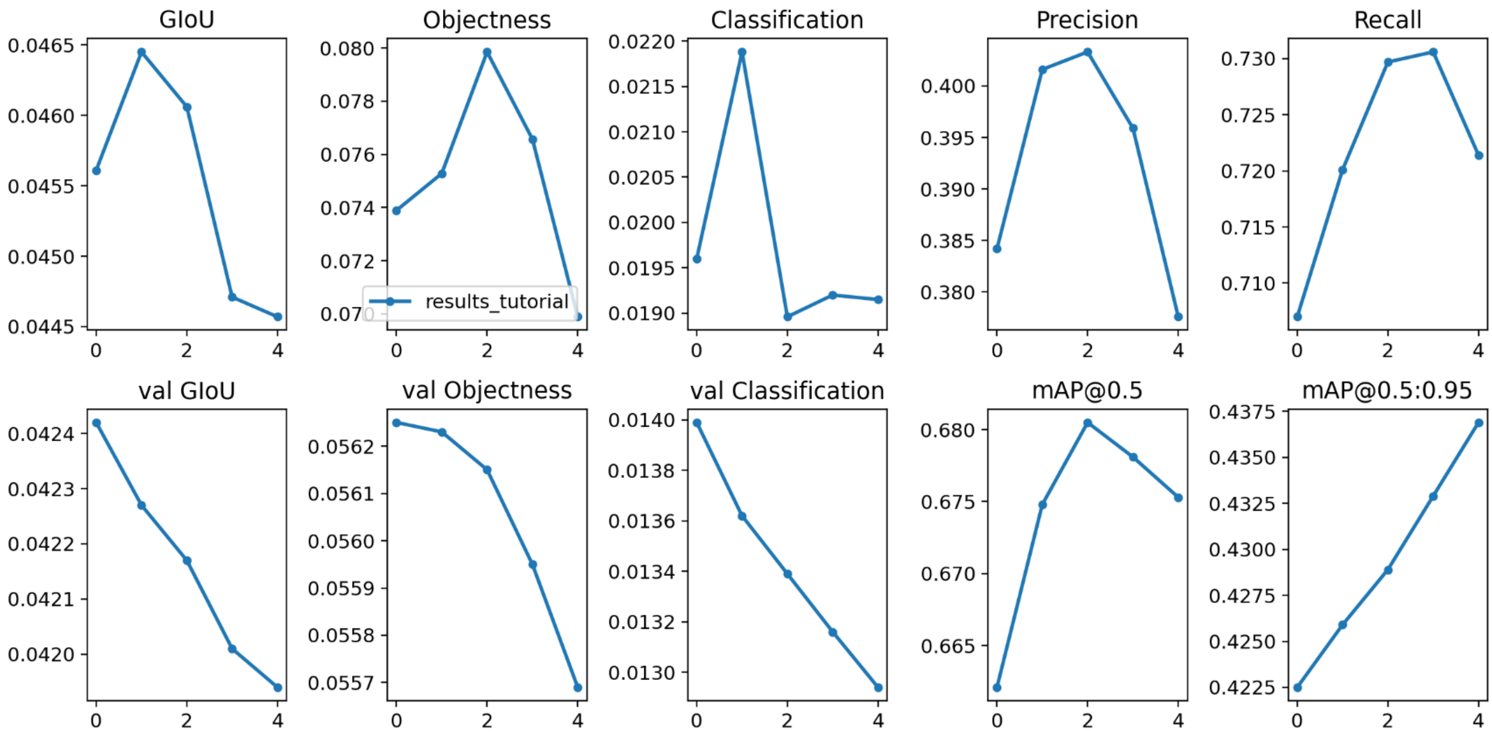
```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.470
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.659
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.515
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.310
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.516
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.610
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.362
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.597
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.656
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.504
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.704
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.787
```

Model Summary:

```
             from  n    params  module                                  arguments
 0             -1  1      3520  models.common.Focus                     [3, 32, 3]
 1             -1  1     18560  models.common.Conv                      [32, 64, 3, 2]
 2             -1  1     20672  models.common.Bottleneck                [64, 64]
 3             -1  1     73984  models.common.Conv                      [64, 128, 3, 2]
 4             -1  1    161152  models.common.BottleneckCSP             [128, 128, 3]
 5             -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
 6             -1  1    641792  models.common.BottleneckCSP             [256, 256, 3]
 7             -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
 8             -1  1    656896  models.common.SPP                       [512, 512, [5, 9, 13]]
 9             -1  1   1905152  models.common.BottleneckCSP             [512, 512, 2]
10             -1  1   1248768  models.common.BottleneckCSP             [512, 512, 1, False]
11             -1  1    130815  torch.nn.modules.conv.Conv2d            [512, 255, 1, 1]
12             -2  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
13        [-1, 6]  1         0  models.common.Concat                    [1]
14             -1  1    197120  models.common.Conv                      [768, 256, 1, 1]
15             -1  1    313088  models.common.BottleneckCSP             [256, 256, 1, False]
16             -1  1     65535  torch.nn.modules.conv.Conv2d            [256, 255, 1, 1]
17             -2  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
18        [-1, 4]  1         0  models.common.Concat                    [1]
19             -1  1     49408  models.common.Conv                      [384, 128, 1, 1]
20             -1  1     78720  models.common.BottleneckCSP             [128, 128, 1, False]
21             -1  1     32895  torch.nn.modules.conv.Conv2d            [128, 255, 1, 1]
22   [-1, 16, 11]  1         0  models.yolo.Detect                      [80, [[10, 13, 16, 30, 33, 23], [30, 61, 62, 45, 59, 119], [116, 90,
odel Summary: 165 layers, 7.07417e+06 parameters, 7.07417e+06 gradients
```

```
     Epoch   gpu_mem      GIoU       obj       cls     total   targets  img_size
       0/4      7.3G   0.04561   0.07389    0.0196    0.1391       217       640: 100% 8/8 [00:15<00:00,  1.90s/it]
               Class    Images   Targets         P         R    mAP@.5  mAP@.5:.95: 100% 8/8 [00:14<00:00,  1.80s/it]
                 all       128       929     0.384     0.707     0.662     0.422

     Epoch   gpu_mem      GIoU       obj       cls     total   targets  img_size
       1/4     11.1G   0.04645   0.07528   0.02188    0.1436       152       640: 100% 8/8 [00:06<00:00,  1.16it/s]
               Class    Images   Targets         P         R    mAP@.5  mAP@.5:.95: 100% 8/8 [00:04<00:00,  1.84it/s]
                 all       128       929     0.402      0.72     0.675     0.426

     Epoch   gpu_mem      GIoU       obj       cls     total   targets  img_size
       2/4     11.1G   0.04606   0.07985   0.01896    0.1449       222       640: 100% 8/8 [00:06<00:00,  1.16it/s]
               Class    Images   Targets         P         R    mAP@.5  mAP@.5:.95: 100% 8/8 [00:04<00:00,  1.94it/s]
                 all       128       929     0.403      0.73     0.681     0.429

     Epoch   gpu_mem      GIoU       obj       cls     total   targets  img_size
       3/4     11.1G   0.04471   0.07657    0.0192    0.1405       215       640: 100% 8/8 [00:06<00:00,  1.15it/s]
               Class    Images   Targets         P         R    mAP@.5  mAP@.5:.95: 100% 8/8 [00:04<00:00,  1.89it/s]
                 all       128       929     0.396     0.731     0.678     0.433

     Epoch   gpu_mem      GIoU       obj       cls     total   targets  img_size
       4/4     11.1G   0.04457   0.06989   0.01915    0.1336       153       640: 100% 8/8 [00:07<00:00,  1.14it/s]
               Class    Images   Targets         P         R    mAP@.5  mAP@.5:.95: 100% 8/8 [00:04<00:00,  1.89it/s]
                 all       128       929     0.378     0.721     0.675     0.437
Optimizer stripped from weights/last_tutorial.pt
5 epochs completed in 0.021 hours.
```

Parameters



GIoU
Objectness
Classification
Precision
Recall
val GIoU
val Objectness
val Classification
mAP@0.5
mAP@0.5:0.95
results_tutorial

Output

Conclusion

YOLO v5 is lightweight and extremely easy to use. YOLO v5 trains quickly, inferences quickly, and performs well.