Project: Real time Weapon Detection

Final Report

Group:
Edvard H
Sivasankar Vennala
Srinivas Rao Polimera
V2.0

Table of Contents

1. Project Description

Today, most of the criminal activities are taken place using handheld arms particularly gun, pistol and revolver. The crime and social offence can be reduced by monitoring such activities and to identify antisocial behaviour so that law enforcement agencies can take appropriate action in early stage. The solution of this problem is to deploy surveillance system or control cameras with automatic handheld gun detection and alert system. In last few years, deep learning has established a landmark in the area of machine learning particularly in object detection, classification and image segmentation (Semantic segmentation and Instance segmentation). Convolutional Neural Network (CNN) achieved best results so far in classical image processing problems such as image segmentation, classification and detection in several applications.

Some other problems that arise during gun detection are noises in gun image, deformation of gun and real time processing requirement. During the transportation and addition of images, noises can occur in that image. Various types of noises can be there like salt and pepper noise, Raleigh noise, Gaussian noise, etc. and these noises can be introduced during varied conditions of transportation and addition of noises.

In this project we present an automatic handheld gun detection system using deep learning techniques particularly YOLO and Detectron2. From the experiments we found that YOLO model training very fast, but the results are not promising and Detectron2 model training little slow compared to YOLO, but it given most promising results.

The best performing model always shows a high potential even in low quality images and provides satisfactory results. So, we decided to build a hybrid model using YOLO and Detectron2 together which gives better speed performance and accuracy, by the leveraging the best aspects of both approaches.

We have used data from the given in below link:
https://sci2s.ugr.es/weapons-detection
The objectives of the project are,

- Learn to how to do build an Object Detection Model
- Use transfer learning to fine-tune a model
- Use different deep learning techniques to see the performance of the models and compare results.
- Create a hybrid approach model to get better performance and accuracy from the final model selection.
- Read different research papers of given domain to obtain the knowledge of advanced models for the given problem.

2. Introduction

A. Defining problem statement

Current surveillance and control systems still require human supervision and intervention. Thus far, previous work has mostly focused on weapon-based detection within infrared data for concealed weapons. By contrast, we are particularly interested in the rapid detection and identification of weapons from images and surveillance data. We reformulate this detection problem into the problem of minimizing false positives and solve it by building the key training dataset guided by the results of a deep Convolutional Neural Networks, then assessing best models using region proposal approach. Hence, we have used YOLO and Detectron2 to build our model. When it comes to object detection, there are two metrics that are important for evaluating their performance: speed and accuracy. When it comes to object detection, one tends to see a trade-off between speed and accuracy (see figure 1). The most promising results are obtained by hybrid approach model trained using YOLO and Detectron2.

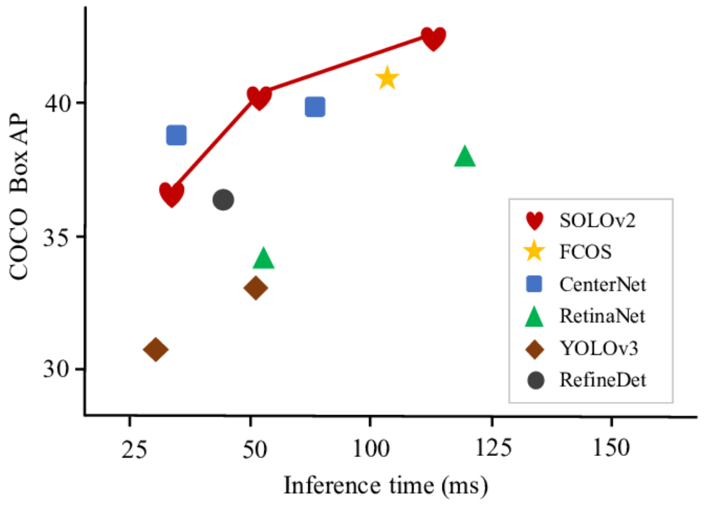


Figure 1: the accuracy and speed trade-off in object detection using different approaches (taken from Wang, Zhang & Kong, 2020)

B. Need of the study/project

With acts of violence causing 1.6 million deaths worldwide every year and terror on the rise, there are unique challenges that make up the core of the weapon and explosives detection systems market, presenting some unique opportunities. In the past, the most prevalent challenge has been the balance of convenience and security, typified by the type of experience shared by travellers in airports. With airports and flights established as desirable targets, individual scans at security checkpoints have become routine and expected. While ensuring that passengers remain safe, these scans slow the flow of traffic through security, exchanging convenience for security.

In public spaces such as concert venues, shopping malls, and schools, the balance often shifts in the other direction, exchanging tight security for convenient, free and direct access. These scenarios present clear challenges that security professionals are forced to address. In many cases shooters carried more than one weapon making the issue of detection all the more important, and while shootings in schools and houses of worship tend to occupy our focus, attacks in offices, retail establishments, and restaurants are more prevalent; detection is needed in all of these spaces.

In addition, new technologies such as 3D printing have allowed those who would commit acts of violence to purchase or create weapons that could defy traditional scans such as metal detectors, making them even more difficult to stop.

C. Understanding business/social opportunity

The world would be a better place if there were no market weapon and explosives detection systems at all. Security professionals around the world would agree that, in a perfect world, public spaces and the people who use them would remain safe and secure. The truth is that we don't live in that place, and the work that we do to advance technology to deliver a safer world is important. By eliminating the need for individual checkpoints while monitoring individuals passing through in real time, we need a active monitoring/detection technology works in real-time to identify threats through machine/deep learning algorithms, helping to identify not only existing weapons, but also learning as new types of weapons become available.

Additionally, for real-time monitoring to be effective – one needs an approach that is both accurate and capable of processing the frames in real time. If the approach is highly accurate on single images but takes longer to process than the frame rate, there will be considerable lag and the potential to detect and flag threats will be also delayed – thereby reducing its efficiency.

3. Preliminary Research on Data

Data which we used for this project collected from Soft Computing and Intelligent Information Systems website. Project scope and focus mainly on the detection of handguns in images and videos using Deep Learning techniques based on CNN (Convolutional Neural Networks).
The training dataset, appropriate for the detection task, contains 3000 images of guns with rich context and Test dataset for both classification and detection. A total of 608 images of which 304 are images of pistols.

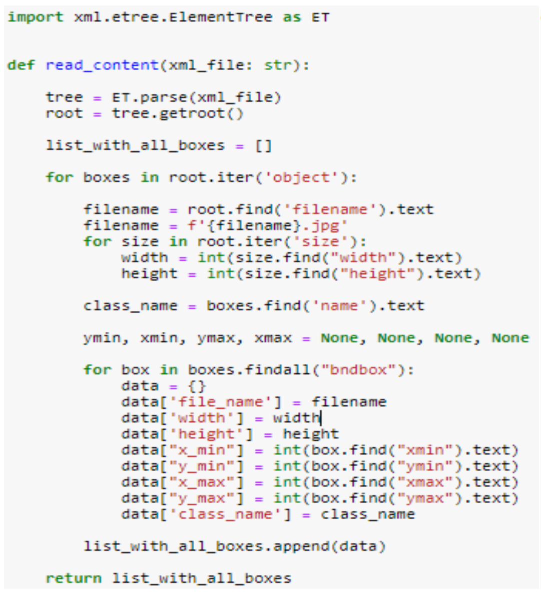
4. Methodology

A. Exploratory Data Analysis

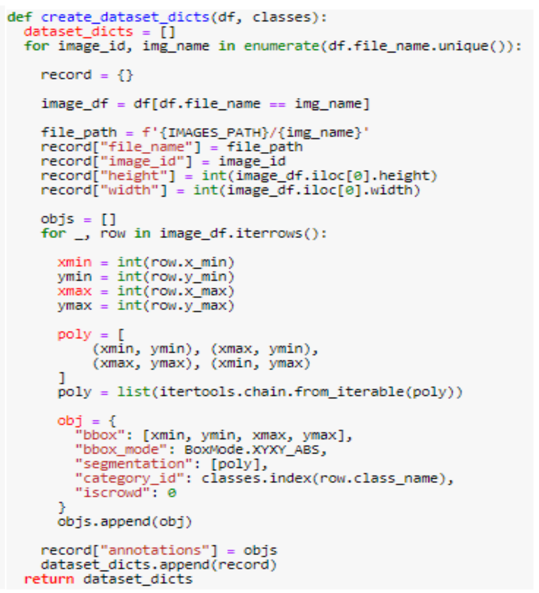
1. Data Pre-processing

The most important part of data science and machine learning is data and data must be neat and clean for models to process it. Data pre-processing methodologies vary with regards to the data types. When it comes to image data, there are other preparation methods. Every state-of-the-art model was built with some assumptions about the data like the size or the number of channels. So, the dataset needs to be resized, or the architecture of the network changed. The image can be resized to get rid of unnecessary information in the background. When these steps are done, data needs to be analysed using various statistical methods to get some insights.
Dataset which we used in this project contains annotations (bounding boxes) data in PASCAL VOC XML format which we need to convert into a shape which supports for YOLO and Detectron2 models.

Detectron2:
We convert every annotation row to a single record with a list of annotations. You might also notice that we're building a polygon that is of the exact same shape as the bounding box. This is required for the image segmentation models in Detectron2. We have to register dataset into the dataset and metadata catalogues of Detectron2 pipeline.

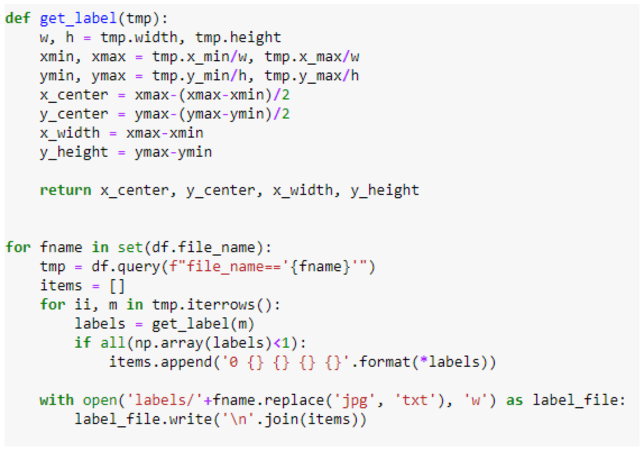


```python
import xml.etree.ElementTree as ET


def read_content(xml_file: str):

    tree = ET.parse(xml_file)
    root = tree.getroot()

    list_with_all_boxes = []

    for boxes in root.iter('object'):

        filename = root.find('filename').text
        filename = f'{filename}.jpg'
        for size in root.iter('size'):
            width = int(size.find("width").text)
            height = int(size.find("height").text)

        class_name = boxes.find('name').text

        ymin, xmin, ymax, xmax = None, None, None, None

        for box in boxes.findall("bndbox"):
            data = {}
            data['file_name'] = filename
            data['width'] = width
            data['height'] = height
            data["x_min"] = int(box.find("xmin").text)
            data["y_min"] = int(box.find("ymin").text)
            data["x_max"] = int(box.find("xmax").text)
            data["y_max"] = int(box.find("ymax").text)
            data['class_name'] = class_name

        list_with_all_boxes.append(data)

    return list_with_all_boxes
```



```python
def create_dataset_dicts(df, classes):
  dataset_dicts = []
  for image_id, img_name in enumerate(df.file_name.unique()):

    record = {}

    image_df = df[df.file_name == img_name]

    file_path = f'{IMAGES_PATH}/{img_name}'
    record["file_name"] = file_path
    record["image_id"] = image_id
    record["height"] = int(image_df.iloc[0].height)
    record["width"] = int(image_df.iloc[0].width)

    objs = []
    for _, row in image_df.iterrows():

      xmin = int(row.x_min)
      ymin = int(row.y_min)
      xmax = int(row.x_max)
      ymax = int(row.y_max)

      poly = [
          (xmin, ymin), (xmax, ymin),
          (xmax, ymax), (xmin, ymax)
      ]
      poly = list(itertools.chain.from_iterable(poly))

      obj = {
        "bbox": [xmin, ymin, xmax, ymax],
        "bbox_mode": BoxMode.XYXY_ABS,
        "segmentation": [poly],
        "category_id": classes.index(row.class_name),
        "iscrowd": 0
      }
      objs.append(obj)

    record["annotations"] = objs
    dataset_dicts.append(record)
  return dataset_dicts
```

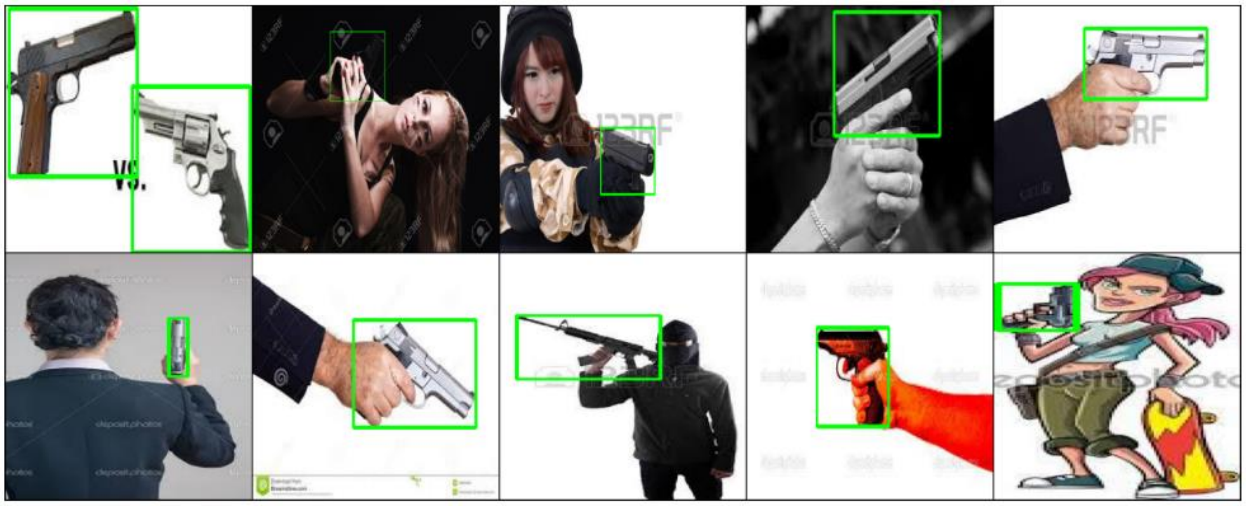
YOLO-tiny:
The images that were preprocessed for Detectron2 were also used for YOLO. The implementation of YOLO used requires a __.names file with a list of the target categories (in our case one: gun) and a text file to be created for each of the annotated images. The text file contains the class index number (in our case always 0 as we only had one class) and bounding box data. Each annotated object in each image is line space separated. These text files were created from the associated XML files of our data. A text file for pointing to the training data is also required.



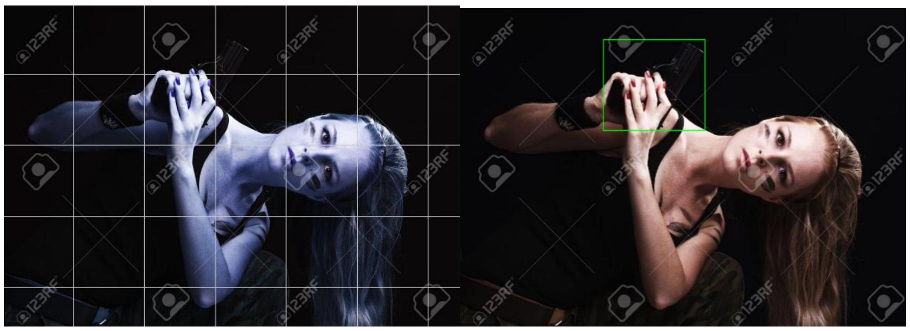
```
def get_label(tmp):
    w, h = tmp.width, tmp.height
    xmin, xmax = tmp.x_min/w, tmp.x_max/w
    ymin, ymax = tmp.y_min/h, tmp.y_max/h
    x_center = xmax-(xmax-xmin)/2
    y_center = ymax-(ymax-ymin)/2
    x_width = xmax-xmin
    y_height = ymax-ymin

    return x_center, y_center, x_width, y_height


for fname in set(df.file_name):
    tmp = df.query(f"file_name=='{fname}'")
    items = []
    for ii, m in tmp.iterrows():
        labels = get_label(m)
        if all(np.array(labels)<1):
            items.append('0 {} {} {} {}'.format(*labels))

    with open('labels/'+fname.replace('jpg', 'txt'), 'w') as label_file:
        label_file.write('\n'.join(items))
```

2. Data Exploration

Now, let us dive deeper into the data. Lets see some sample images with annotations on handgun detection from training set.

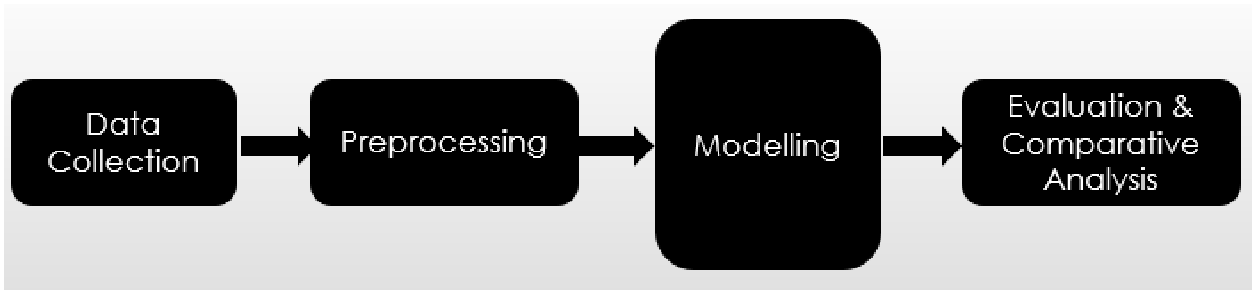


As we can see in above sample images with bounding boxes which we use to train our models.



5. Methods and models

The main goal is predicting weapons in each image or video streaming. They do so by predicting bounding boxes around areas of the weapon. Samples without bounding boxes are negative and contain no definitive evidence of weapons. Samples with bounding boxes indicate evidence of weapons. When making predictions, there should predict as many bounding boxes as necessary, in the format: confidence x-min, y-min, width height. We use different kind of Convolutional Neural Network models to solve this this problem.
We will follow the following data pipeline to collect data, pre-processing, model and evaluation process.



A. Pre-trained models used

We used following pre trained model further to check the model performance and further improvements.

1. **Detectron2**
2. **YOLO**
3. **Hybrid Model (Combination of YOLO and Detectron2)**

Detectron2:

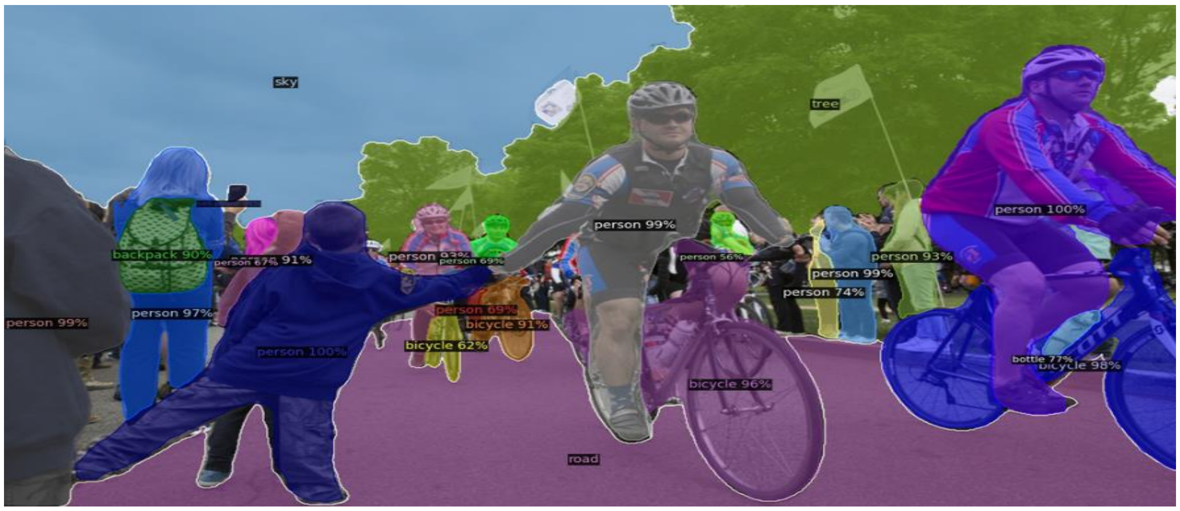
Detectron2 is FAIR's next-generation platform for object detection and segmentation which is developed and released by Facebook AI Research (FAIR) as an open-source project. It is a second generation of the library as the first Detectron was written in Caffe2 and then with the maskrcnn-benchmark reimplemented in PyTorch 1.0. Detectron2 is a ground-up rewrite and extension of the previous effort using PyTorch.

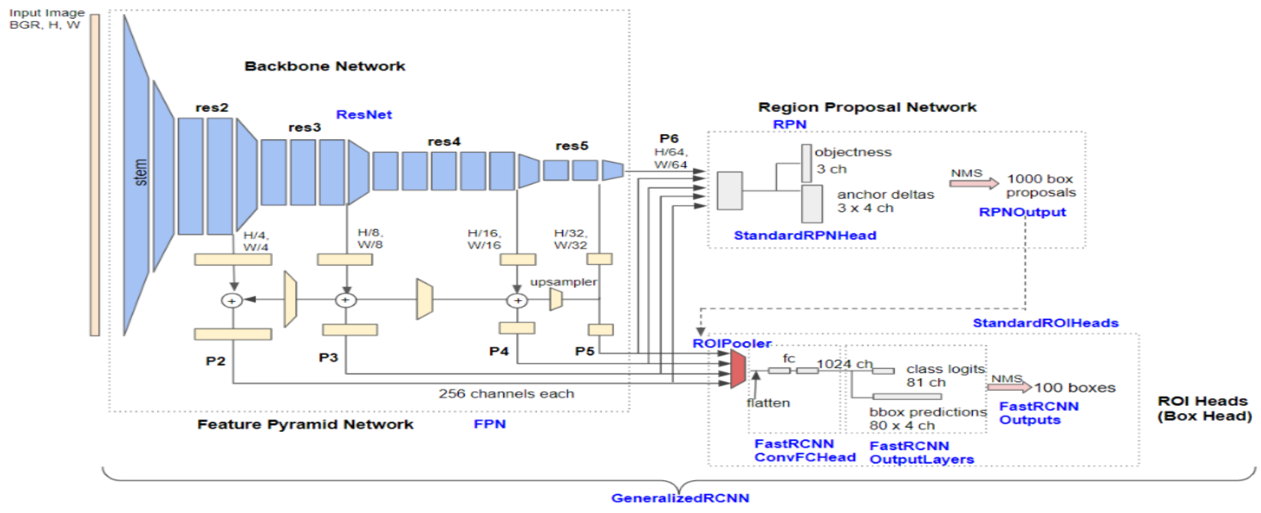
Detectron2 beyond state-of-the-art object detection algorithms includes numerous models like instance segmentation, panoptic segmentation, pose estimation, DensePose, Trident Net.

Detectron2 was built to support rapid implementation and evaluation of novel computer vision research. It includes implementations for the following object detection algorithms:

- Mask R-CNN
- RetinaNet
- Faster R-CNN
- RPN
- Fast R-CNN
- TensorMask
- PointRend
- DensePose

and more...





For detectron2 model block – see appendix A.

Detectron2 Validation and Testing Performance:

From training and validation detectron2 achieved the following performance.

(Validation) Model Evaluation Summary:

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.599
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.907
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.647
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.318
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.336
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.687
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.559
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.701
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.714
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.488
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.570
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.767
[32m[05/22 11:12:52 d2.evaluation.coco_evaluation]: [0mEvaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 59.939 | 90.688 | 64.741 | 31.836 | 33.586 | 68.749 |
```

Test Accuracy:

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.609
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.886
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.685
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.188
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.321
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.677
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.559
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.690
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.707
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.314
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.516
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.760
[04/30 11:26:32 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 60.889 | 88.623 | 68.536 | 18.776 | 32.133 | 67.735 |
```

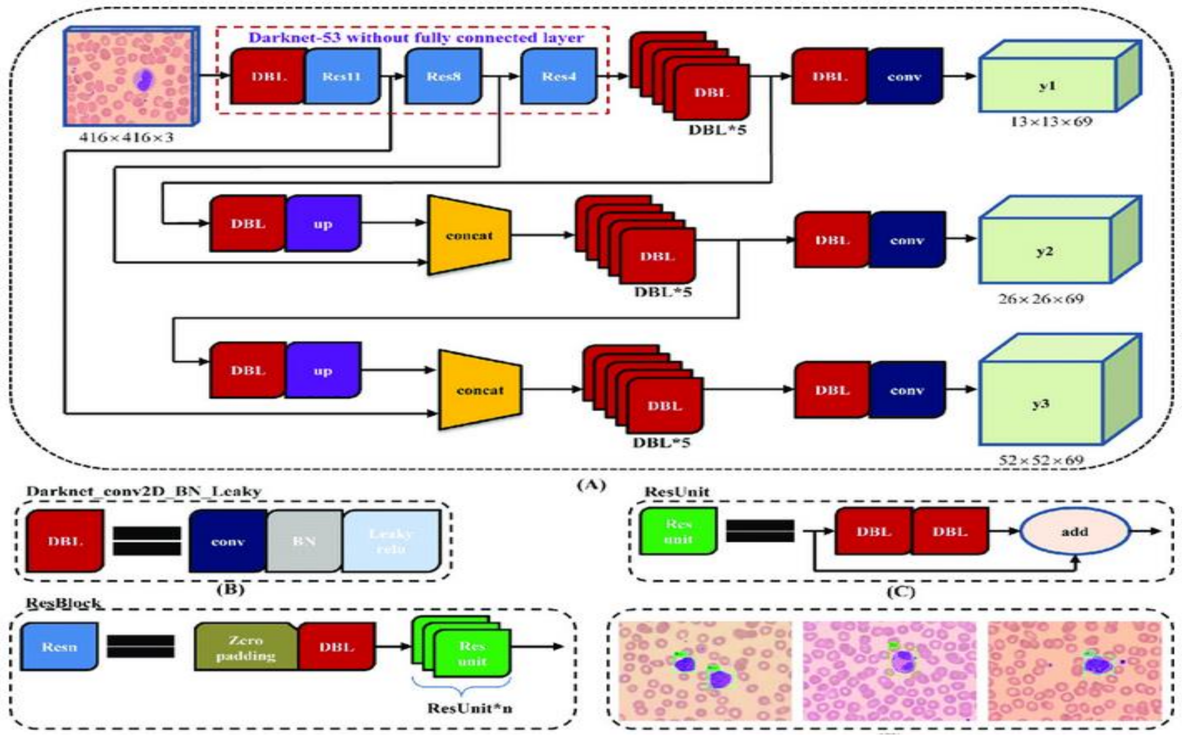
Model summary shows an mAP@0.5 of 90.688 after 1500 epochs for detectron2 and an mAP@0.5 if 88.623 for the testing data. Mean inference time per image was 197 ms for the testing set.

Detectron2 example predictions**:**

YOLO:

YOLO (You Only Look Once), is a network for object detection. The object detection task consists in determining the location on the image where certain objects are present, as well as classifying those objects. Previous methods for this, like R-CNN and its variations, used a pipeline to perform this task in multiple steps. This can be slow to run and hard to optimize, because each individual component must be trained separately. YOLO, does it all with a single neural network.

The newer architecture boasts of residual skip connections, and up sampling. The most salient feature of YOLO v3 is that it makes detections at three different scales. YOLO is a fully convolutional network and its eventual output is generated by applying a 1 x 1 kernel on a feature map. In YOLO v3, the detection is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network. YOLO v3 makes prediction at three scales, which are precisely given by down sampling the dimensions of the input image by 32, 16 and 8 respectively.



For YOLO-tiny model block see appendix B.

<u>YOLO-tiny Validation and Testing Performance:</u>

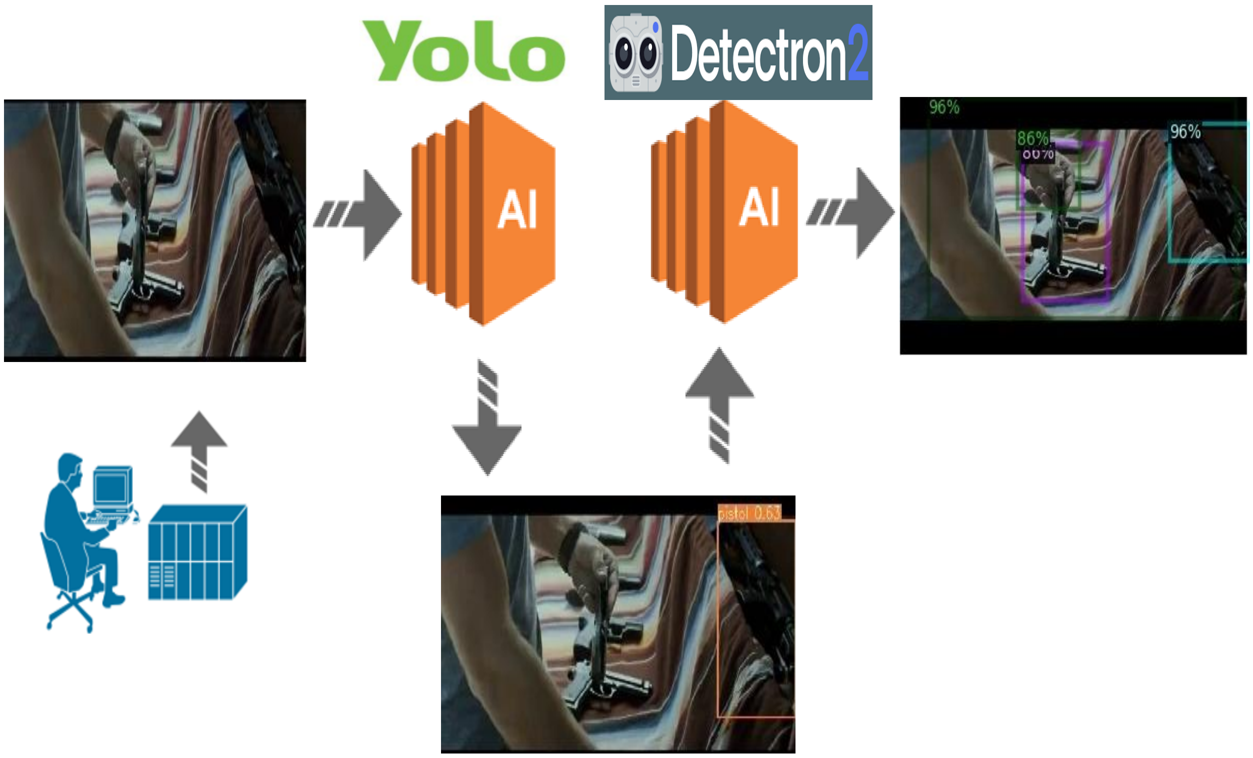
The output from YOLO-tiny is not as verbose as Detectron2. However, from the training and validation YOLO-tiny achieved an mAP@0.5 of 79.9 after 300 epochs and an mAP@0.5 of 85.3 for the testing data. Mean inference time per image for the testing set was 18ms.

YOLO-tiny example predictions:

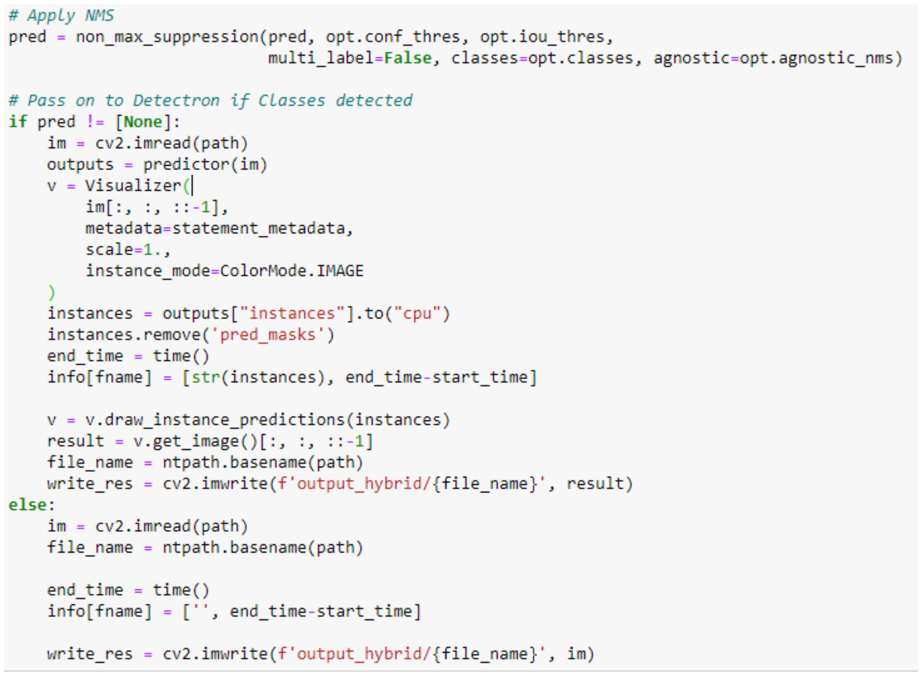
B. Hybrid Approach

When comparing the performance of YOLO-tiny and Detectron2 with fast-RCNN, while detectron2 performed better on precision (88.6 vs. 85.3) YOLO-tiny was much faster (18ms vs 197ms). We proposed that by combining the two and using YOLO-tiny as a filter would yield high accuracies and improve the speed of detection2 for yolo on average. This is because detectron2 iterates over every image thousands of times, regardless if there is a weapon or not, whereas YOLO only iterates once (hence the name – You Only Look Once).

In order to improve the accuracy in terms of both IoU and likelihood of weapon detection lowering YOLO-tiny's confidence threshold would mean that some of the frames/images where YOLO suspects there is a weapon will be passed on to detectron2 for inference. This might increase the likelihood of false positives but it also means that edge cases from YOLO will be confirmed or rejected by detectron2.



Consider figure 2, when YOLO uses it's default confidence threshold of 0.3 it does not generate a prediction for gun as confidence prediction for gun is lower. By using a lower threshold it generates a prediction for which the ground truth does not reflect the predicted grounding box. By passing it on to Detectron2, we get a more accurate bounding box and prediction.



```
# Apply NMS
pred = non_max_suppression(pred, opt.conf_thres, opt.iou_thres,
                           multi_label=False, classes=opt.classes, agnostic=opt.agnostic_nms)

# Pass on to Detectron if Classes detected
if pred != [None]:
    im = cv2.imread(path)
    outputs = predictor(im)
    v = Visualizer(
        im[:, :, ::-1],
        metadata=statement_metadata,
        scale=1.,
        instance_mode=ColorMode.IMAGE
    )
    instances = outputs["instances"].to("cpu")
    instances.remove('pred_masks')
    end_time = time()
    info[fname] = [str(instances), end_time-start_time]

    v = v.draw_instance_predictions(instances)
    result = v.get_image()[:, :, ::-1]
    file_name = ntpath.basename(path)
    write_res = cv2.imwrite(f'output_hybrid/{file_name}', result)
else:
    im = cv2.imread(path)
    file_name = ntpath.basename(path)

    end_time = time()
    info[fname] = ['', end_time-start_time]

    write_res = cv2.imwrite(f'output_hybrid/{file_name}', im)
```

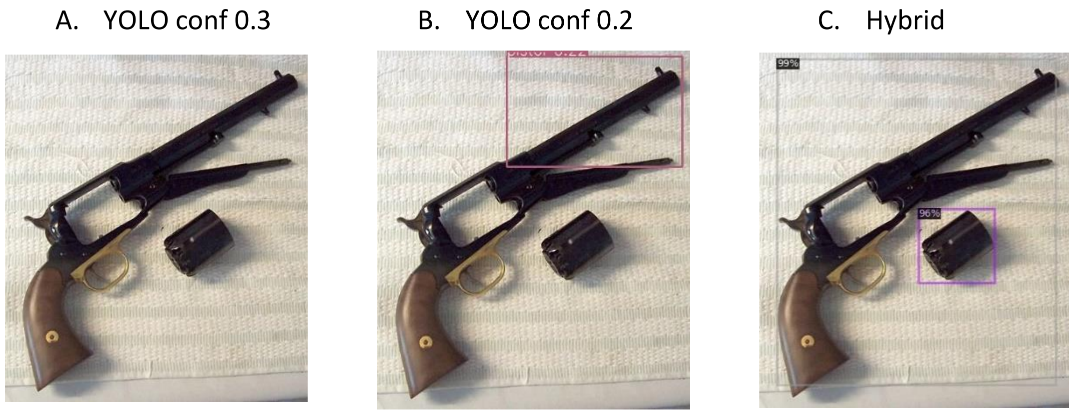


Figure 2 – YOLO vs Hybrid approach: A) demonstrates YOLO's performance using a confidence threshold of 0.3. **B)** YOLO using a confidence threshold of 0.2. **C)** YOLO with a confidence threshold of 0.2 and passed on to detectron2.

If an image contains no weapon and YOLO "confident" that there is no weapon it will not be passed on to detectron2. Consider 300 images of weapons and 300 images of no weapons. Detectron2 would take 118.2 seconds to do inference on all images whereas YOLO would take 10.8 seconds. However, YOLO would not be as accurate as detectron2. By combining the two approaches we should see accuracies more closely resemble detectron2's performance while the total inference time should take approximately 64.6 seconds. This is because detectron2 would not need to iterate over every single image thousands of times, including the images where there are no guns (unless YOLO generates a false positive).

In order to test this, we fed the images through YOLO-tiny and if a prediction was generated, the image was then passed on to detectron2 for detection.

Model	m**AP**	**Mean Inference Time** (ms)
Detectron2 (Fast-RCNN)	88.6	197
YOLO-Tiny	85.3	18
Hybrid	90.2	49

For evaluation, mAP was calculated using only 150 images with weapons, whereas inference time was calculated using those same images and 150 images without weapons. We wanted to use all 300 images for both metrics, however, there technical issues in getting Detectron2 to perform evaluation on un-annotated data. Therefore, conclusions about improved accuracies and precision between the 3 approaches should be treated with caution, however, the mean inference time can be compared across all three.

When it comes to speed of processing, we can clearly see the benefit of potentially using YOLO as a filter for detectron2 as the mean inference time went from 197 ms to 49 ms. When comparing YOLO-tiny to hybrid, clearly YOLO-tiny was faster (18 ms), however, we would hope that hybrid would demonstrate improved accuracy when compared to YOLO. However, as mentioned we cannot at this time draw conclusions from said comparison.

6. Weapons Detection in Videos

In order to illustrate the benefit of this using the hybrid approach on videos we tested the performance of each model and hybrid on a video. The video was taken from YouTube and contains a 25 seconds scene of an ongoing illegal weapons sale. The video is not annotated and so we cannot provide precision metrics, however, we will provide examples of edge cases and compare and contrast each methods prediction and time performance.

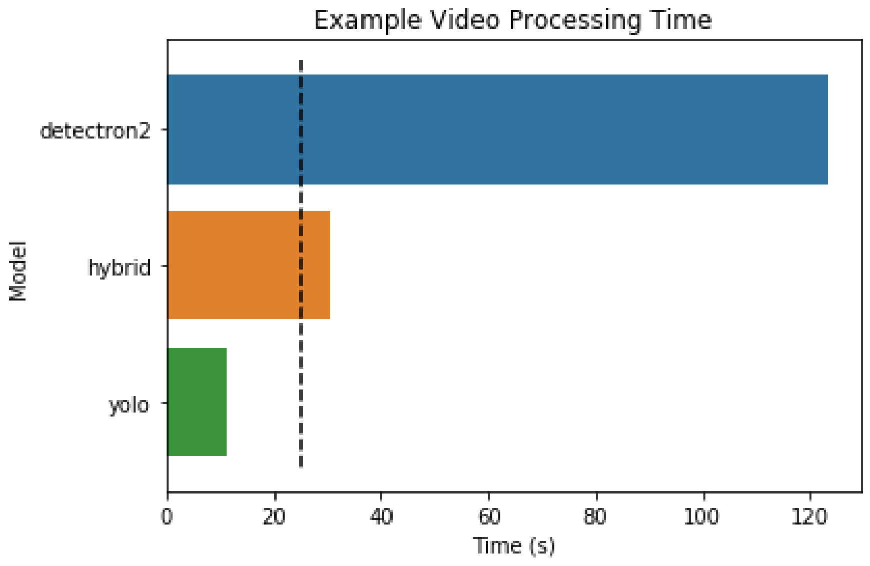


Figure 3: Total inference time for a 25 second video. Detectron2 took the longest to process the video (123.3 s) followed by hybrid (30.6 s – with conf. thres. = 0.2) and then YOLO-tiny (11.3 s). Reference line indicates the duration of the video (25 s)

Inference processing time:

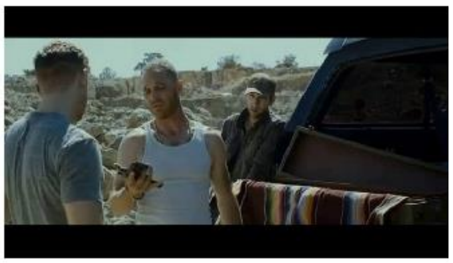
As can be seen from figure 3, Detectron2 was exceptionally slow when it came to analysing the video. The advantage of speed that YOLO has becomes increasingly obvious in this case. While hybrid took a bit longer than the duration of the video, it is to be expected as most of the frames did include an image of a gun and so many of the frames were subsequently passed on to detectron2. However, in regard to precision, while we have no metrics to compare their performance, we can demonstrate the benefits by using a few edge cases to demonstrate the benefit of using the hybrid approach.

Edge Cases in Video Inference

As seen in figures 4 and 5, YOLO struggles with certain edge cases while the hybrid approach performs better. For example, in figure 4A YOLO does not detect the pistol when it at an angle, however, with a conf. thresh. of 0.2 and passed on to detectron2 the hybrid approach performs better in these cases.

Figure 4. Video performance: Angled Gun

A. YOLO



B. Hybrid

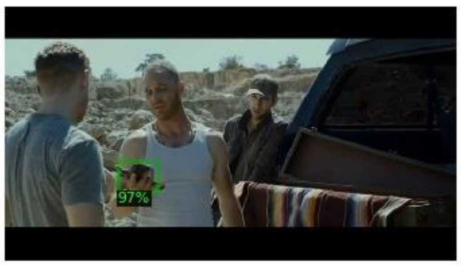
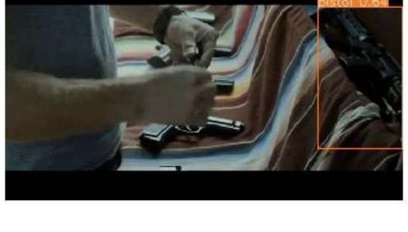
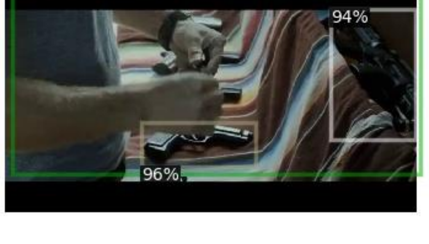


Figure 5. Video performance: Obscured Gun

A. YOLO



B. Hybrid



Similarly, in figure 5 when the pistol is partly obscured by another object (in this case a hand) YOLO is not able to detect the pistol (figure 5A) whereas the hybrid approach does (figure 5B). However, we should note that during the video processing detectron2 in general did seem to have issues with regards to generating strange false positives. Occasionally it would predict that the whole frame contained a gun. This might be due to only using 1 class (weapon) and therefore the model struggles to converge.

7. Possible Improvements and recommendations

Documenting possible improvements for future attempts of this problem. This was done due to time restrictions of project timelines.

There are several ways we could have improved performance. To name a few:

- Using different augmentation parameters
- Train for more epochs (due to limited computational power not able train more epochs now)
- Optimizing the hypermeters using Grid/Random Search approach
- Using other pretrained model weights etc
- Try Other Image segmentation models which are recent segmentation approaches which gives faster and very good results
- Sampling the data to get the equal amount of dataset
- Use more classes for detection (e.g. differentiate between knife, rifle, pistol) to help detectron2's model converge better
- Calculate mAP using annotated and un-annotated data to better compare the speed and accuracy of the 3 approaches

8. Summary

As demonstrated, when processing real time detection in videos inference speed becomes a critical aspect for being able to apply detection on the fly. Using state-of-the art methods, while showing improved accuracies, can be slow – they can additionally benefit from a hybrid approach (as demonstrated here). By using YOLO to filter frames to be later analyzed we were able retain the best aspects of both approaches (i.e. speed and accuracy).

Not only can this approach be used to process live data, but it can also be used in other ways. For example, in research collecting data is a non-trivial task. By using this approach, one could scrape video streaming platforms for relevant data to be used in further improving these methods.

This project gave us a lot of new concepts to digest and a lot of experimenting with code. Overall, it was a really good experience and helped us improve fast over a short period of time. Adequate results were achieved through experimenting on different models and machine learning techniques. We could have done a better job in regards to understanding how Detectron2's data loader worked. To sum it up, it would be fair to say that deep learning technologies have opened paths to attempt to solve complex problems. This was a challenging and exciting problem and we will certainly experiment more and try new ideas in order to get better performance.

9. Reference List

Wang, Xinlong & Zhang, Rufeng & Kong, Tao & Li, Lei & Shen, Chunhua. (2020). SOLOv2: Dynamic, Faster and Stronger.

https://github.com/facebookresearch/detectron2

https://github.com/ultralytics/yolov3

https://sci2s.ugr.es/weapons-detection

https://medium.com/@tont/recognizing-firearms-from-images-and-videos-in-real-time-with-deep-learning-and-computer-vision-661498f45278

10. Appendix

A. Detectron2: Model Block

```
GeneralizedRCNN(
  (backbone): FPN(
    (fpn_lateral2): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral3): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral4): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral5): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (top_block): LastLevelMaxPool()
    (bottom_up): ResNet(
      (stem): BasicStem(
        (conv1): Conv2d(
          3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False
          (norm): FrozenBatchNorm2d(num_features=64, eps=1e-05)
        )
      )
      (res2): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv1): Conv2d(
            64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv3): Conv2d(
            256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv3): Conv2d(
            256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
        )
        (2): BottleneckBlock(
          (conv1): Conv2d(
            256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv2): Conv2d(
            256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
          (conv3): Conv2d(
            256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
          )
        )
      )
      (res3): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv1): Conv2d(
            256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv2): Conv2d(
            512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv3): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv2): Conv2d(
            512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv3): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
        )
        (2): BottleneckBlock(
          (conv1): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv2): Conv2d(
            512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv3): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
        )
        (3): BottleneckBlock(
          (conv1): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv2): Conv2d(
            512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
          (conv3): Conv2d(
            512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=512, eps=1e-05)
          )
        )
      )
      (res4): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv1): Conv2d(
            512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv2): Conv2d(
            1024, 1024, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv3): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv2): Conv2d(
            1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv3): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
        )
        (2): BottleneckBlock(
          (conv1): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv2): Conv2d(
            1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv3): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
        )
        (3): BottleneckBlock(
          (conv1): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv2): Conv2d(
            1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
          (conv3): Conv2d(
            1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
          )
        )
```

```
(4): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(5): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(6): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(7): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(8): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (9): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (10): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (11): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (12): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (13): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (14): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (15): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (16): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (17): BottleneckBlock(
    (conv1): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv2): Conv2d(
      1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
    (conv3): Conv2d(
      1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
    )
  )
  (18): BottleneckBlock(
    (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(19): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(20): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(21): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
)
(22): BottleneckBlock(
  (conv1): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv2): Conv2d(
    1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
  )
  (conv3): Conv2d(
    1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
    (norm): FrozenBatchNorm2d(num_features=1024, eps=1e-05)
```

```
          )
        )
      )
      (res5): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
            1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv1): Conv2d(
            1024, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv2): Conv2d(
            2048, 2048, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv3): Conv2d(
            2048, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
        )
        (1): BottleneckBlock(
          (conv1): Conv2d(
            2048, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv2): Conv2d(
            2048, 2048, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv3): Conv2d(
            2048, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
        )
        (2): BottleneckBlock(
          (conv1): Conv2d(
            2048, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv2): Conv2d(
            2048, 2048, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
          (conv3): Conv2d(
            2048, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
            (norm): FrozenBatchNorm2d(num_features=2048, eps=1e-05)
          )
        )
      )
    )
  )
  (proposal_generator): RPN(
    (anchor_generator): DefaultAnchorGenerator(
      (cell_anchors): BufferList()
    )
    (rpn_head): StandardRPNHead(
      (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (objectness_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
      (anchor_deltas): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (roi_heads): StandardROIHeads(
    (box_pooler): ROIPooler(
      (level_poolers): ModuleList(
        (0): ROIAlign(output_size=(7, 7), spatial_scale=0.25, sampling_ratio=0, aligned=True)
        (1): ROIAlign(output_size=(7, 7), spatial_scale=0.125, sampling_ratio=0, aligned=True)
        (2): ROIAlign(output_size=(7, 7), spatial_scale=0.0625, sampling_ratio=0, aligned=True)
        (3): ROIAlign(output_size=(7, 7), spatial_scale=0.03125, sampling_ratio=0, aligned=True)
      )
    )
    (box_head): FastRCNNConvFCHead(
      (fc1): Linear(in_features=12544, out_features=1024, bias=True)
      (fc2): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNOutputLayers(
      (cls_score): Linear(in_features=1024, out_features=2, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=4, bias=True)
    )
    (mask_pooler): ROIPooler(
      (level_poolers): ModuleList(
        (0): ROIAlign(output_size=(14, 14), spatial_scale=0.25, sampling_ratio=0, aligned=True)
        (1): ROIAlign(output_size=(14, 14), spatial_scale=0.125, sampling_ratio=0, aligned=True)
        (2): ROIAlign(output_size=(14, 14), spatial_scale=0.0625, sampling_ratio=0, aligned=True)
        (3): ROIAlign(output_size=(14, 14), spatial_scale=0.03125, sampling_ratio=0, aligned=True)
      )
    )
    (mask_head): MaskRCNNConvUpsampleHead(
      (mask_fcn1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (mask_fcn2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (mask_fcn3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (mask_fcn4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (deconv): ConvTranspose2d(256, 256, kernel_size=(2, 2), stride=(2, 2))
      (predictor): Conv2d(256, 1, kernel_size=(1, 1), stride=(1, 1))
    )
  )
 )
```

B. YOLO-tiny: Model Block

```
[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
```

```
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

###########

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
```

```
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear



[yolo]
mask = 6,7,8
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  373,326
classes=1
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear

[yolo]
mask = 3,4,5
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  373,326
classes=1
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```

```
[route]
layers = -3

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 6

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  373,326
classes=1
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```