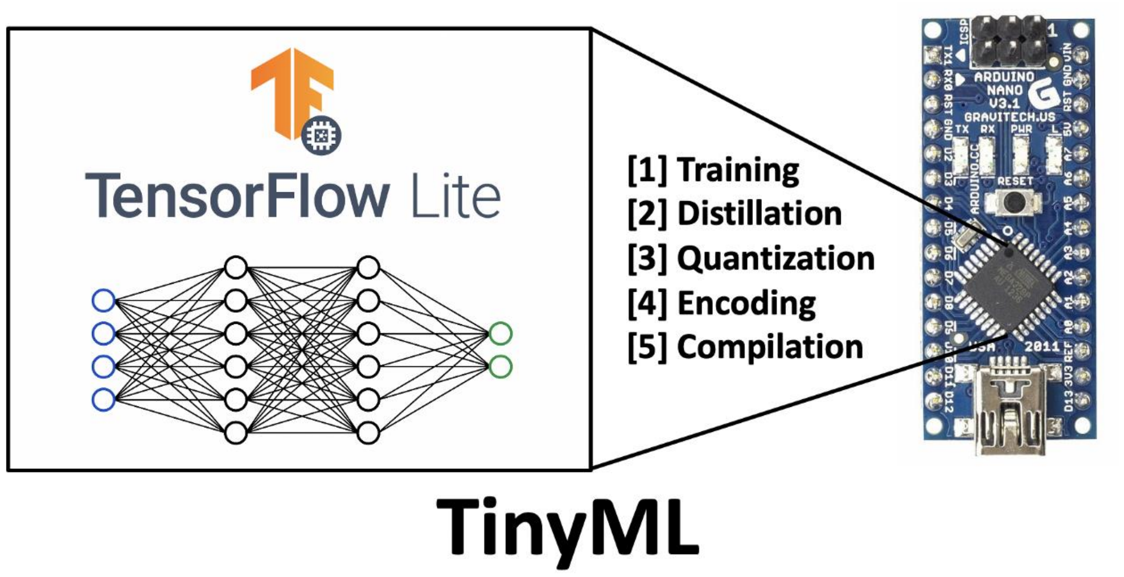
A TECHNICAL OVERVIEW OF MODEL COMPRESSION

A Report by
Akshay Dongare

TABLE OF CONTENTS



INTRODUCTION

What is TinyML?

- TinyML is the intersection of Machine Learning and Embedded Systems that explores ways to enable machine learning models to run on small, low-powered devices like microcontrollers.
- Under powered hardware does not have the liberty to leverage great memory and speed during making predictions using the deployed model (inference stage of machine learning).
- This is where TinyML comes in. It enables low-latency, low power and low bandwidth model inference at edge devices like microcontrollers and mobile phones.

Why TinyML?

What is the need for TinyML?

- As time progresses, Deep Neural Networks are showcasing a trend of containing increasingly many parameters.
- This leads to larger model sizes and greater inference times.

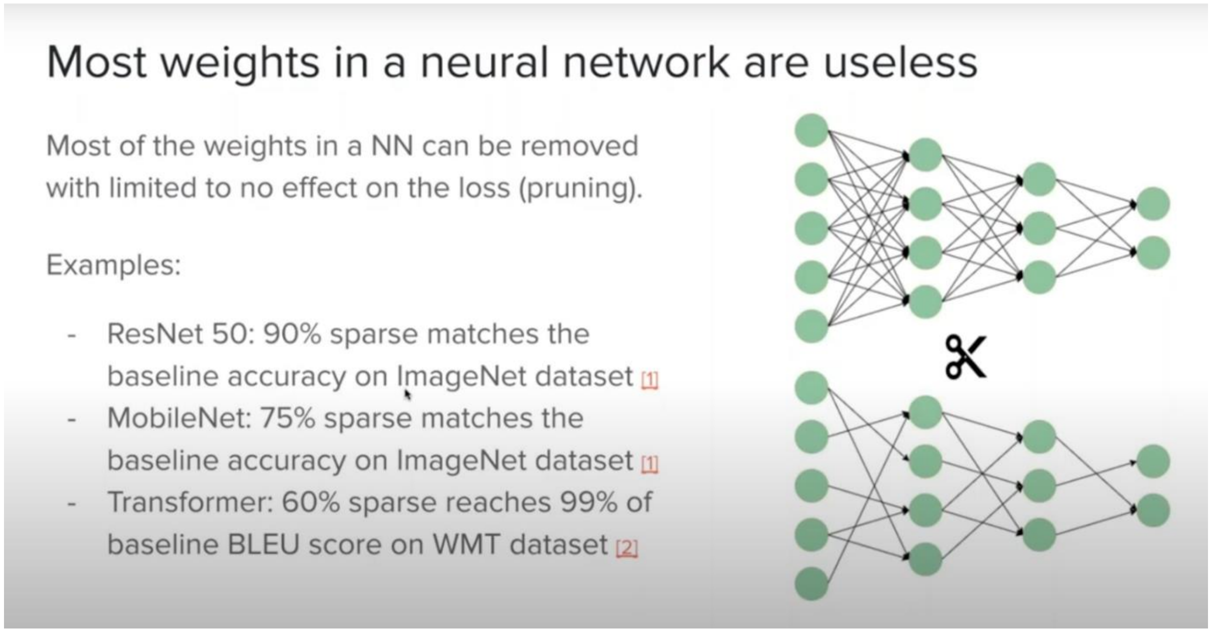
- But research has led to the conclusion that, when it comes to real world applications of Machine Learning, "*Bigger isn't always better*".
- Observing the surge of mobile devices and the need of ML applications to be run on them, TinyML has become an integral part of the effort to solve real life problems.
- For example, the Google voice assistant can detect words with a model just 14 kilobytes in size. That is, small enough to run on a microcontroller.
- In this way, TinyML provides frameworks that enable us to run these models on-the-edge ensuring data privacy and low latency.

How TinyML?

How is TinyML actually implemented?

Model compression

- Model compression is the technique of deploying state-of-the-art deep networks in devices with low power and resources without compromising on the model's accuracy.
- Compressing or reducing in size and/or latency means the model has fewer and smaller parameters and requires lesser RAM.
- The concept of model compression is based on the basic idea that, in a Deep Neural Network, not all of the connections have any significant effect on the output of the network; thus, rendering them useless and redundant.



Most weights in a neural network are useless

Most of the weights in a NN can be removed with limited to no effect on the loss (pruning).

Examples:

- ResNet 50: 90% sparse matches the baseline accuracy on ImageNet dataset [1]
- MobileNet: 75% sparse matches the baseline accuracy on ImageNet dataset [1]
- Transformer: 60% sparse reaches 99% of baseline BLEU score on WMT dataset [2]

MODEL COMPRESSION TECHNIQUES

Pruning

- Deep Neural Networks tend to be over parameterized.
- That is, multiple features convey almost the same information and are inconsequential in the large scheme of things.
- Thus, we can remove some connections between the neurons or sometimes the whole neuron, channel or filter from a trained network. This process is called Pruning.
- Pruning can be classified into unstructured and structured pruning. In unstructured pruning, individual weights or neurons are removed, and in structured pruning, entire channels or filters are taken out.

Quantization

- Unlike Pruning, we do not remove weights or connection in Quantization. Instead, we decrease the weights' size.
- The weights' size can be reduced by reducing the precision of the stored weight values. For example, from Single-precision floating-point format called FP32 (occupies 32 bits of memory) to Integer of 8 bits (INT8).
- The output contains a smaller range of values compared to the input without losing much information in the process.

Selective attention

- Only the objects or elements of interest are focused while the background and other elements are discarded.
- This technique requires the addition of a selective attention network upon the existing AI model.

Low-rank factorization

- This process uses matrix or tensor decomposition to estimate useful parameters.
- A weight matrix with greater dimension and rank can be replaced with smaller dimension matrices through factorization.

Knowledge distillation

- It is an indirect way of compressing a model where an existing larger model, called teacher, trains smaller models called students.
- The goal is to have the same distribution in the student model as available in the teacher model.
- Here, the loss function is minimized during the transfer of knowledge from teacher to student.

PROBLEM STATEMENT

How does combining different compression techniques affect accuracy and final model size?

Approach

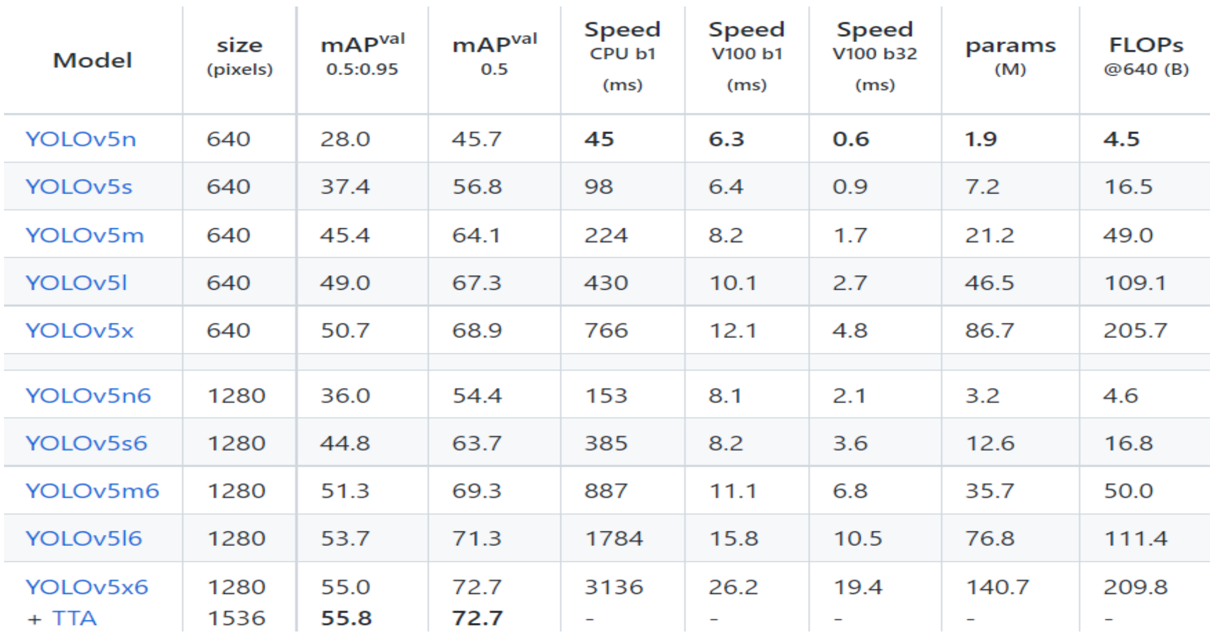
- We will explore how applying certain compression techniques affects
  1. Accuracy
  2. Model Size
  3. Inference Time
- We will also explore whether a certain combination of these techniques, provides a better result.

Yolo (You Only Look Once)

- Developed by Joseph Redmon, YOLO is a series of state-of-the-art Deep Learning models for performing object detection.

YOLOv5

- YOLOv5 🚀 is a family of compound-scaled object detection models trained on the COCO dataset, and includes simple functionality for Test Time Augmentation (TTA), model ensembling, hyperparameter evolution, and export to ONNX, CoreML and TFLite.
- YOLOv5 contains several models such as YOLOv5n, YOLOv5s, YOLOv5l, YOLOv5x etc.



Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	36.0	54.4	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.8	63.7	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.3	69.3	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.7	71.3	1784	15.8	10.5	76.8	111.4
YOLOv5x6	1280	55.0	72.7	3136	26.2	19.4	140.7	209.8
+ TTA	1536	**55.8**	**72.7**	-	-	-	-	-

- These different pre-trained checkpoints are based on the size and speed of the model.
- For benchmarking the performance of this model, we will validate it on the COCO Dataset.

COCO (Common Objects in Context)

- COCO is a large-scale object detection, segmentation, and captioning dataset.
- It is developed by Microsoft.

Evaluation Metrics:

1. Mean Average Precision(mAP):
    - Mean Average Precision(mAP) is a metric used to evaluate object detection models such as Fast R-CNN, YOLO, Mask R-CNN, etc.
    - The mean of average precision (AP) values is calculated over recall values from 0 to 1.

$$mAP = \frac{1}{n}\sum_{k=1}^{k=n} AP_k$$
$$AP_k = \text{the AP of class } k$$
$$n = \text{the number of classes}$$

mAP multi-class formula

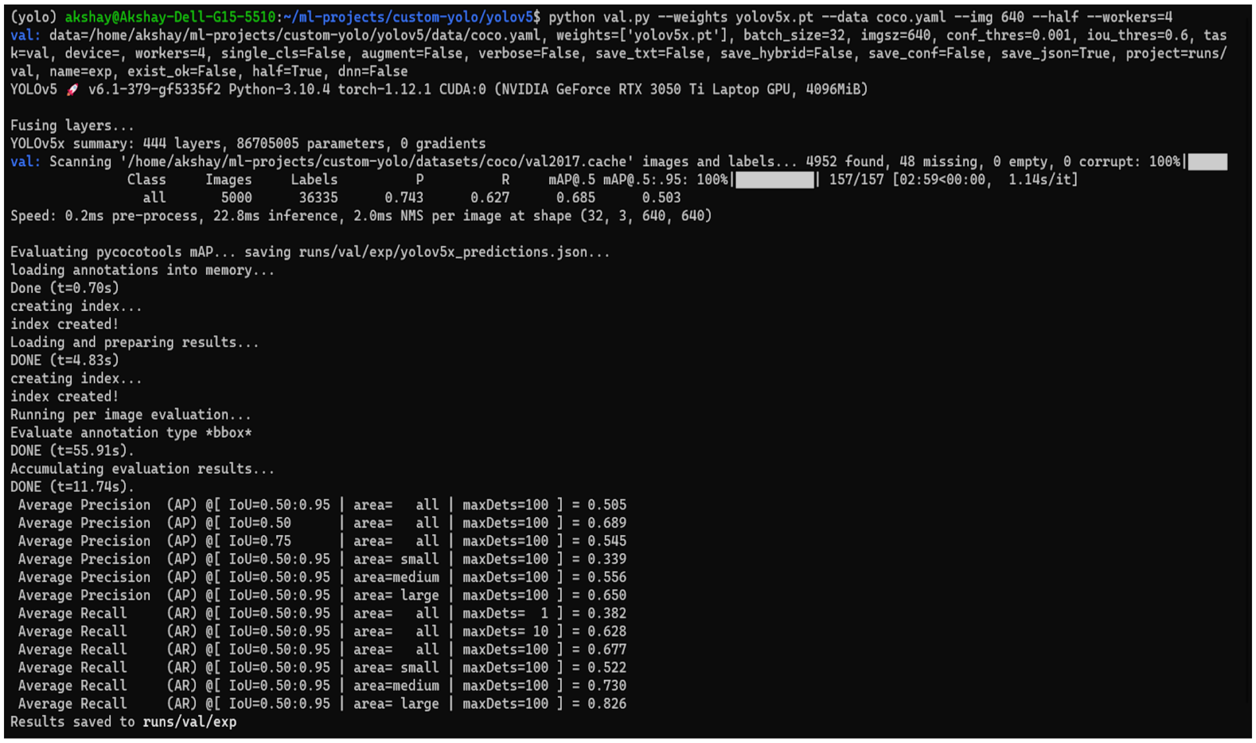
2. Intersection over Union (IoU):
    - Intersection over Union indicates the overlap of the predicted bounding box coordinates to the ground truth box.
    - Higher IoU indicates the predicted bounding box coordinates closely resembles the ground truth box coordinates.

3. COCO mAP:
    - AP is calculated for the IoU threshold of 0.5 for each class.
    - Calculate the precision at every recall value (0 to 1 with a step size of 0.01), then it is repeated for IoU thresholds of 0.55,0.60,…,0.95.
    - Average is taken over all the 80 classes and all the 10 thresholds.

Benchmarking on COCO dataset

- First, we will validate YOLOv5x (the largest available model) on the COCO dataset.

- <u>Output:</u>

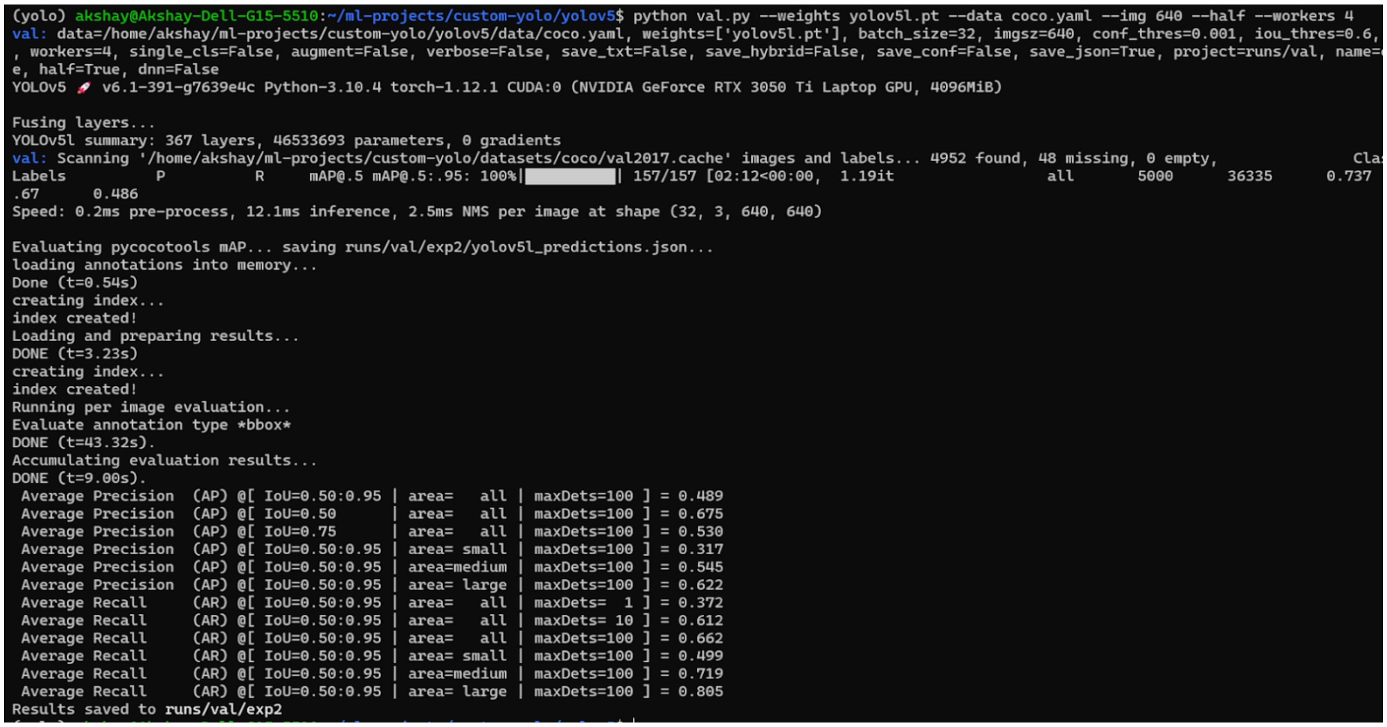


```
(yolo) akshay@Akshay-Dell-G15-5510:~/ml-projects/custom-yolo/yolov5$ python val.py --weights yolov5x.pt --data coco.yaml --img 640 --half --workers=4
val: data=/home/akshay/ml-projects/custom-yolo/yolov5/data/coco.yaml, weights=['yolov5x.pt'], batch_size=32, imgsz=640, conf_thres=0.001, iou_thres=0.6, tas
k=val, device=, workers=4, single_cls=False, augment=False, verbose=False, save_txt=False, save_hybrid=False, save_conf=False, save_json=True, project=runs/
val, name=exp, exist_ok=False, half=True, dnn=False
YOLOv5 🚀 v6.1-379-gf5335f2 Python-3.10.4 torch-1.12.1 CUDA:0 (NVIDIA GeForce RTX 3050 Ti Laptop GPU, 4096MiB)

Fusing layers...
YOLOv5x summary: 444 layers, 86705005 parameters, 0 gradients
val: Scanning '/home/akshay/ml-projects/custom-yolo/datasets/coco/val2017.cache' images and labels... 4952 found, 48 missing, 0 empty, 0 corrupt: 100%|█████
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100%|██████████| 157/157 [02:59<00:00,  1.14s/it]
                 all       5000      36335      0.743      0.627      0.685      0.503
Speed: 0.2ms pre-process, 22.8ms inference, 2.0ms NMS per image at shape (32, 3, 640, 640)

Evaluating pycocotools mAP... saving runs/val/exp/yolov5x_predictions.json...
loading annotations into memory...
Done (t=0.70s)
creating index...
index created!
Loading and preparing results...
DONE (t=4.83s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=55.91s).
Accumulating evaluation results...
DONE (t=11.74s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.505
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.689
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.545
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.339
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.556
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.650
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.382
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.628
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.677
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.522
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.730
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.826
Results saved to runs/val/exp
```

- We can see that we get a mAP of 0.505 and a mAR of 0.382

Benchmarking YOLOv5l model on COCO dataset:

- <u>Output:</u>

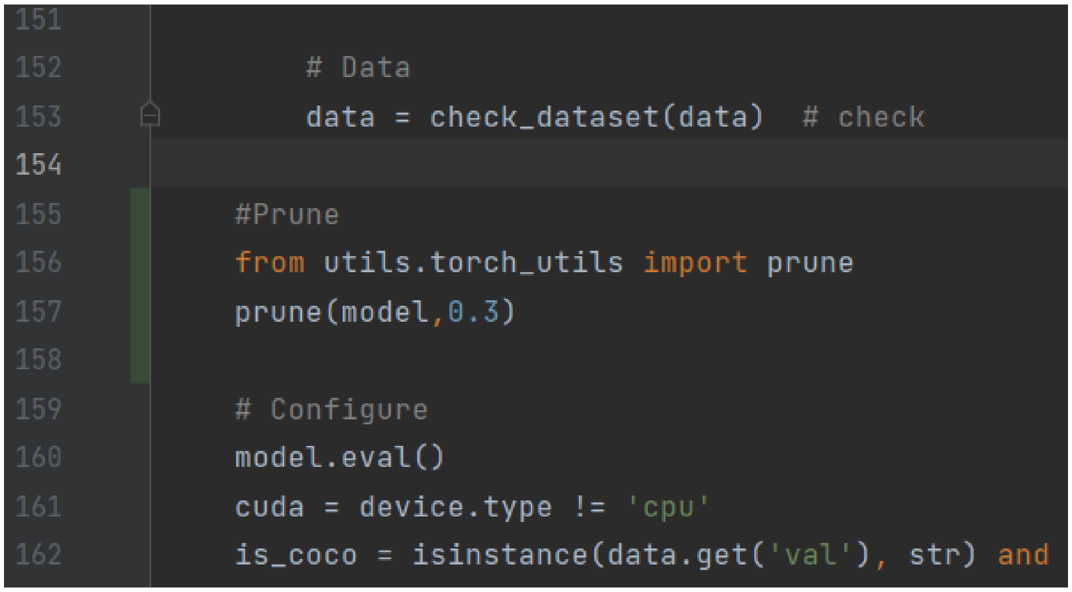


```
(yolo) akshay@Akshay-Dell-G15-5510:~/ml-projects/custom-yolo/yolov5$ python val.py --weights yolov5l.pt --data coco.yaml --img 640 --half --workers 4
val: data=/home/akshay/ml-projects/custom-yolo/yolov5/data/coco.yaml, weights=['yolov5l.pt'], batch_size=32, imgsz=640, conf_thres=0.001, iou_thres=0.6,
, workers=4, single_cls=False, augment=False, verbose=False, save_txt=False, save_hybrid=False, save_conf=False, save_json=True, project=runs/val, name=
e, half=True, dnn=False
YOLOv5 🚀 v6.1-391-g7639e4c Python-3.10.4 torch-1.12.1 CUDA:0 (NVIDIA GeForce RTX 3050 Ti Laptop GPU, 4096MiB)

Fusing layers...
YOLOv5l summary: 367 layers, 46533693 parameters, 0 gradients
val: Scanning '/home/akshay/ml-projects/custom-yolo/datasets/coco/val2017.cache' images and labels... 4952 found, 48 missing, 0 empty,            Cla
Labels         P          R     mAP@.5 mAP@.5:.95: 100%|██████████| 157/157 [02:12<00:00,  1.19it                 all       5000      36335      0.737
.67      0.486
Speed: 0.2ms pre-process, 12.1ms inference, 2.5ms NMS per image at shape (32, 3, 640, 640)

Evaluating pycocotools mAP... saving runs/val/exp2/yolov5l_predictions.json...
loading annotations into memory...
Done (t=0.54s)
creating index...
index created!
Loading and preparing results...
DONE (t=3.23s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=43.32s).
Accumulating evaluation results...
DONE (t=9.00s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.489
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.675
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.530
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.317
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.545
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.622
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.372
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.612
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.662
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.499
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.719
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.805
Results saved to runs/val/exp2
```

- We can see that we get a mAP of 0.489 and a mAR of 0.372
- Note that the slightly smaller values of mAP and mAR in comparison to YOLOv5x are due to the fact that the size of YOLOv5l model is smaller than YOLOv5x.
- That is, it has lesser total parameters.

Pruning the Models

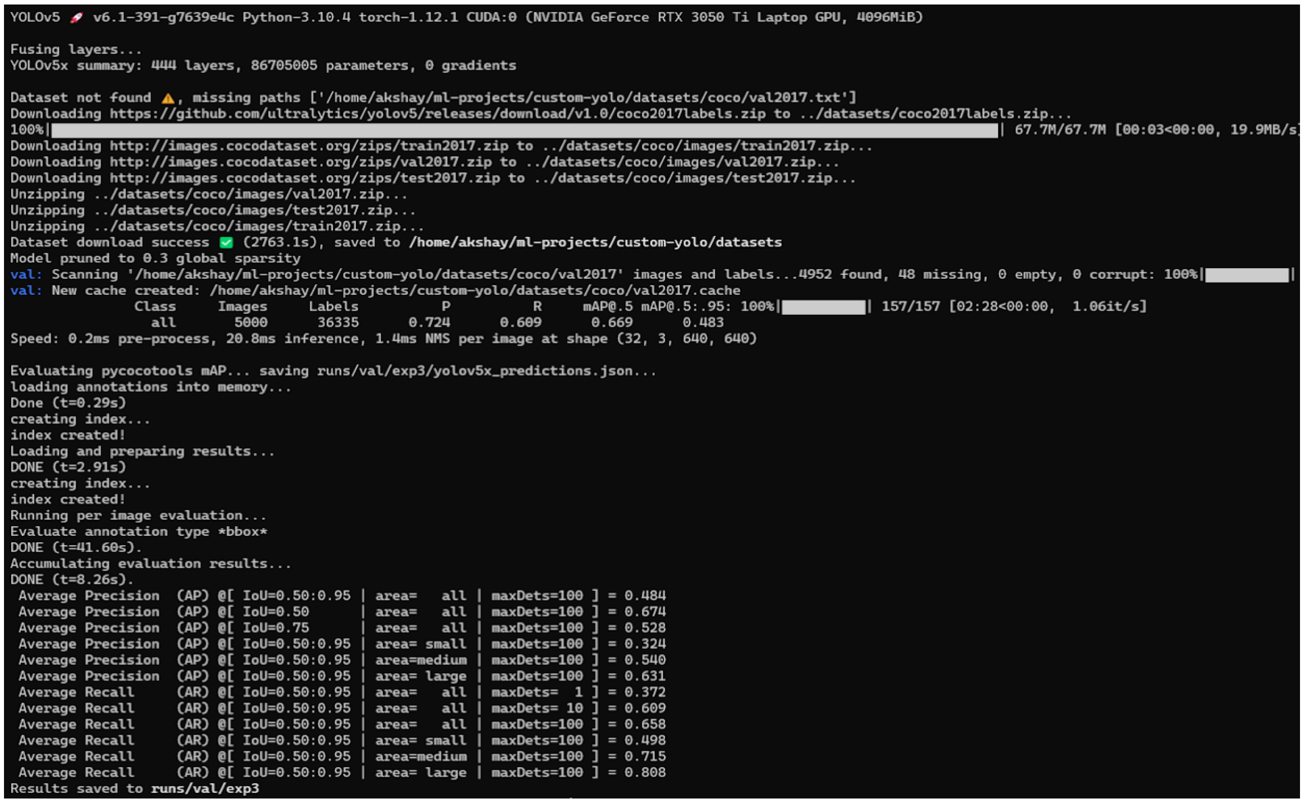
- We will now prune the models to 30% global sparsity.
- This can be done using the `torch_utils.prune()` command and adding the required code in the val.py file like so,



```
        # Data
        data = check_dataset(data)  # check


    #Prune
    from utils.torch_utils import prune
    prune(model,0.3)

    # Configure
    model.eval()
    cuda = device.type != 'cpu'
    is_coco = isinstance(data.get('val'), str) and
```

YOLOv5x Pruned to 0.3 Sparsity:

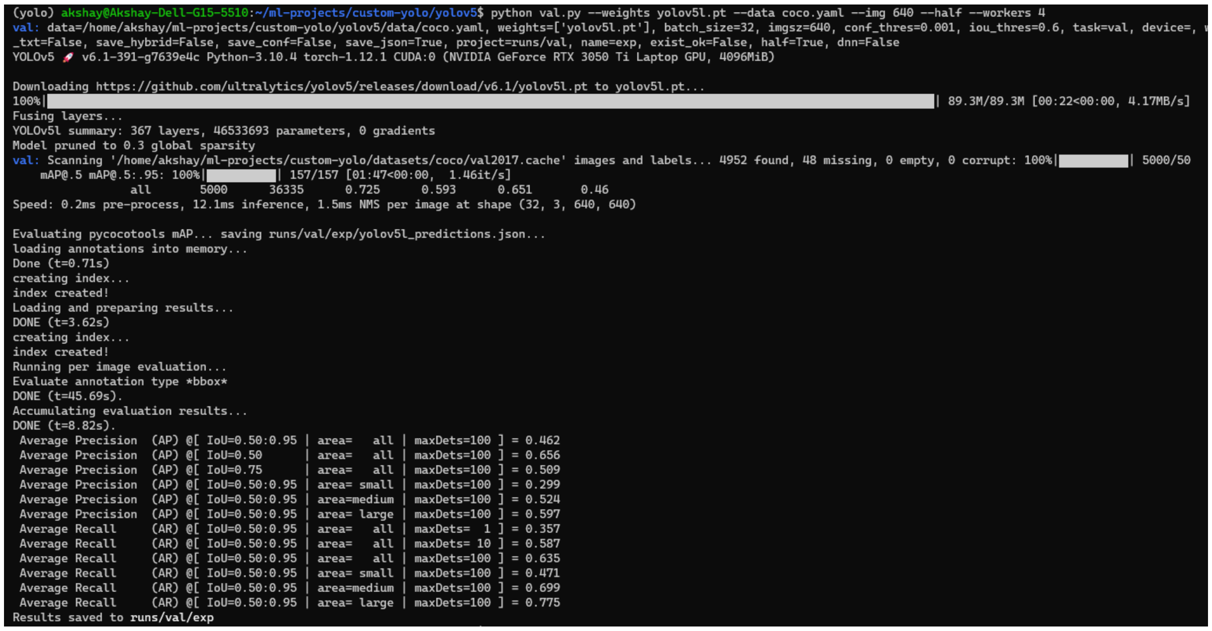
OUTPUT:



```
YOLOv5 🚀 v6.1-391-g7639e4c Python-3.10.4 torch-1.12.1 CUDA:0 (NVIDIA GeForce RTX 3050 Ti Laptop GPU, 4096MiB)

Fusing layers...
YOLOv5x summary: 444 layers, 86705005 parameters, 0 gradients

Dataset not found ⚠, missing paths ['/home/akshay/ml-projects/custom-yolo/datasets/coco/val2017.txt']
Downloading https://github.com/ultralytics/yolov5/releases/download/v1.0/coco2017labels.zip to ../datasets/coco2017labels.zip...
100%|████████████████████████████████████████████████████████████████████████| 67.7M/67.7M [00:03<00:00, 19.9MB/s
Downloading http://images.cocodataset.org/zips/train2017.zip to ../datasets/coco/images/train2017.zip...
Downloading http://images.cocodataset.org/zips/val2017.zip to ../datasets/coco/images/val2017.zip...
Downloading http://images.cocodataset.org/zips/test2017.zip to ../datasets/coco/images/test2017.zip...
Unzipping ../datasets/coco/images/val2017.zip...
Unzipping ../datasets/coco/images/test2017.zip...
Unzipping ../datasets/coco/images/train2017.zip...
Dataset download success ✅ (2763.1s), saved to /home/akshay/ml-projects/custom-yolo/datasets
Model pruned to 0.3 global sparsity
val: Scanning '/home/akshay/ml-projects/custom-yolo/datasets/coco/val2017' images and labels...4952 found, 48 missing, 0 empty, 0 corrupt: 100%|██████████|
val: New cache created: /home/akshay/ml-projects/custom-yolo/datasets/coco/val2017.cache
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100%|██████████| 157/157 [02:28<00:00,  1.06it/s]
                 all       5000      36335      0.724      0.609      0.669      0.483
Speed: 0.2ms pre-process, 20.8ms inference, 1.4ms NMS per image at shape (32, 3, 640, 640)

Evaluating pycocotools mAP... saving runs/val/exp3/yolov5x_predictions.json...
loading annotations into memory...
Done (t=0.29s)
creating index...
index created!
Loading and preparing results...
DONE (t=2.91s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=41.60s).
Accumulating evaluation results...
DONE (t=8.26s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.484
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.674
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.528
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.324
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.540
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.631
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.372
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.609
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.658
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.498
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.715
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.808
Results saved to runs/val/exp3
```

YOLOv5l Pruned to 0.3 Sparsity:

<u>OUTPUT:</u>



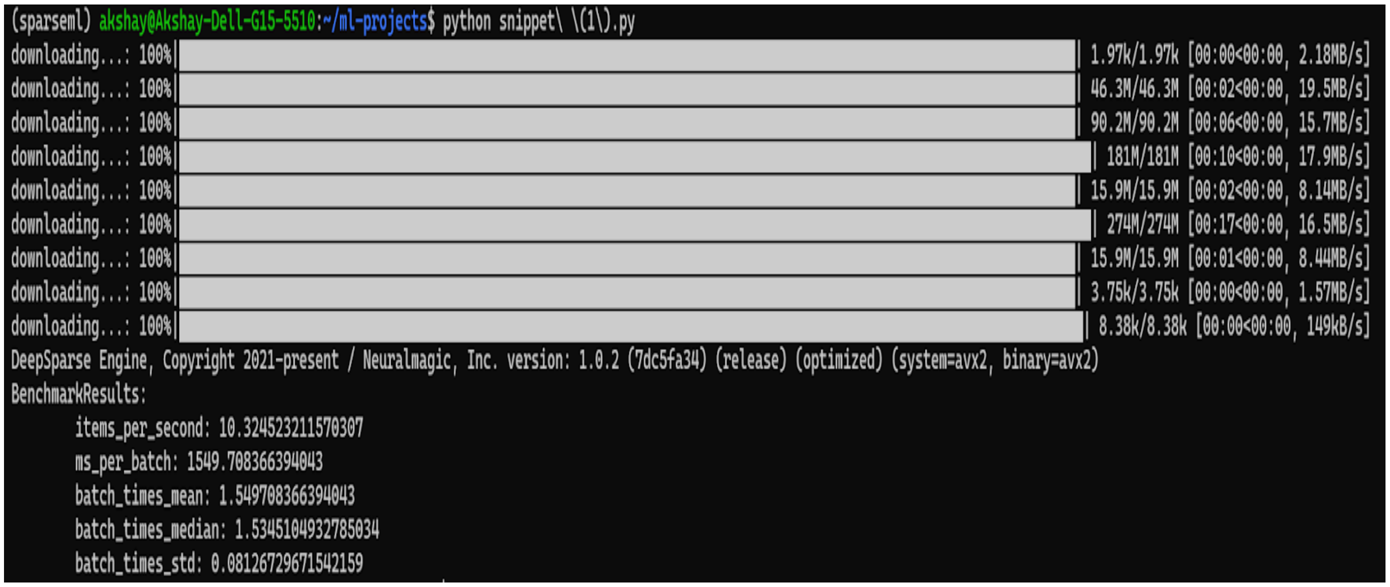
```
(yolo) akshay@Akshay-Dell-G15-5510:~/ml-projects/custom-yolo/yolov5$ python val.py --weights yolov5l.pt --data coco.yaml --img 640 --half --workers 4
val: data=/home/akshay/ml-projects/custom-yolo/yolov5/data/coco.yaml, weights=['yolov5l.pt'], batch_size=32, imgsz=640, conf_thres=0.001, iou_thres=0.6, task=val, device=, w
_txt=False, save_hybrid=False, save_conf=False, save_json=True, project=runs/val, name=exp, exist_ok=False, half=True, dnn=False
YOLOv5 🚀 v6.1-391-g7639e4c Python-3.10.4 torch-1.12.1 CUDA:0 (NVIDIA GeForce RTX 3050 Ti Laptop GPU, 4096MiB)

Downloading https://github.com/ultralytics/yolov5/releases/download/v6.1/yolov5l.pt to yolov5l.pt...
100%|████████████████████████████████████████████████████████████████████████| 89.3M/89.3M [00:22<00:00, 4.17MB/s]
Fusing layers...
YOLOv5l summary: 367 layers, 46533693 parameters, 0 gradients
Model pruned to 0.3 global sparsity
val: Scanning '/home/akshay/ml-projects/custom-yolo/datasets/coco/val2017.cache' images and labels... 4952 found, 48 missing, 0 empty, 0 corrupt: 100%|██████████| 5000/50
    mAP@.5 mAP@.5:.95: 100%|██████████| 157/157 [01:47<00:00,  1.46it/s]
                 all       5000      36335      0.725      0.593      0.651       0.46
Speed: 0.2ms pre-process, 12.1ms inference, 1.5ms NMS per image at shape (32, 3, 640, 640)

Evaluating pycocotools mAP... saving runs/val/exp/yolov5l_predictions.json...
loading annotations into memory...
Done (t=0.71s)
creating index...
index created!
Loading and preparing results...
DONE (t=3.62s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=45.69s).
Accumulating evaluation results...
DONE (t=8.82s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.462
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.656
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.509
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.299
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.524
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.597
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.357
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.587
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.635
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.471
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.699
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.775
Results saved to runs/val/exp
```

- In the results, we can observe that we have achieved a sparsity of 30% in our model after pruning.
- This means that 30% of the model's weight parameters in `nn.Conv2d` layers are equal to 0.
- Inference time is essentially unchanged, while the model's AP and AR scores are slightly reduced.

Quantizing our Pruned Model:

(YOLOv5l Quant + Pruned)

Output:



```
(sparseml) akshay@Akshay-Dell-G15-5510:~/ml-projects$ python snippet\ \(1\).py
downloading...: 100%|████████████████████████████████████| 1.97k/1.97k [00:00<00:00, 2.18MB/s]
downloading...: 100%|████████████████████████████████████| 46.3M/46.3M [00:02<00:00, 19.5MB/s]
downloading...: 100%|████████████████████████████████████| 90.2M/90.2M [00:06<00:00, 15.7MB/s]
downloading...: 100%|█████████████████████████████████████| 181M/181M [00:10<00:00, 17.9MB/s]
downloading...: 100%|████████████████████████████████████| 15.9M/15.9M [00:02<00:00, 8.14MB/s]
downloading...: 100%|█████████████████████████████████████| 274M/274M [00:17<00:00, 16.5MB/s]
downloading...: 100%|████████████████████████████████████| 15.9M/15.9M [00:01<00:00, 8.44MB/s]
downloading...: 100%|████████████████████████████████████| 3.75k/3.75k [00:00<00:00, 1.57MB/s]
downloading...: 100%|█████████████████████████████████████| 8.38k/8.38k [00:00<00:00, 149kB/s]
DeepSparse Engine, Copyright 2021-present / Neuralmagic, Inc. version: 1.0.2 (7dc5fa34) (release) (optimized) (system=avx2, binary=avx2)
BenchmarkResults:
        items_per_second: 10.324523211570307
        ms_per_batch: 1549.708366394043
        batch_times_mean: 1.549708366394043
        batch_times_median: 1.5345104932785034
        batch_times_std: 0.08126729671542159
```

Benchmark Results:

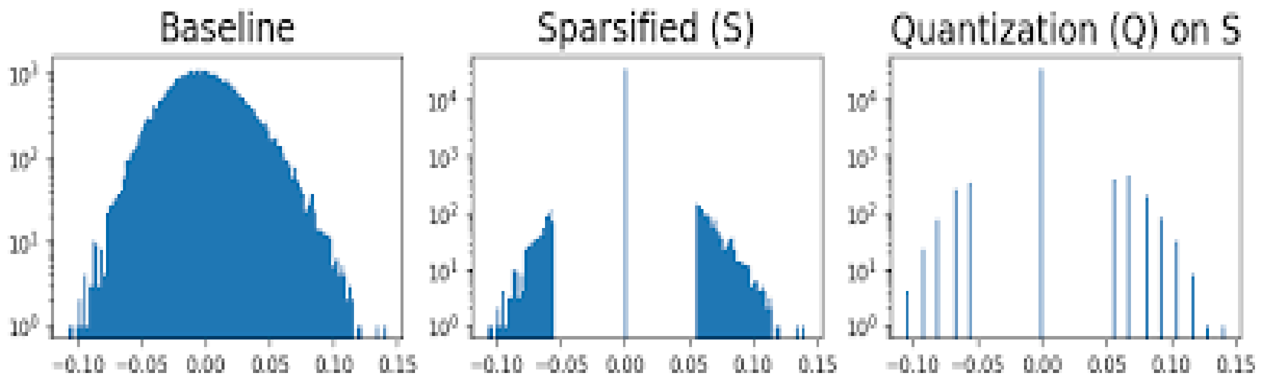
items_per_second: 10.324523211570307

ms_per_batch: 1549.708366394043

batch_times_mean: 1.549708366394043

batch_times_median: 1.5345104932785034

batch_times_std: 0.08126729671542159



CONCLUSION

By applying both pruning and INT8 quantization to the model, we are able to achieve 10x faster inference performance on CPUs and 12x smaller model file sizes.

FUTURE SCOPE

- We can experiment with different optimization libraries like TensorFlow Model Optimization Python API
- We can deploy quantized as well as pruned model on edge devices in real time

REFERENCES AND CITATIONS

- You Only Look Once: Unified, Real-Time Object Detection, 2015: https://doi.org/10.48550/arXiv.1506.02640
- COCO Dataset: https://cocodataset.org/#home
- YOLOv5 Github: https://github.com/ultralytics/yolov5
- Deep Sparse: https://github.com/neuralmagic/deepsparse
- TensorFlow Model Optimization Python API: https://www.tensorflow.org/model_optimization/api_docs/python/tfmot