Players vs Referees - object detection

University of Verona
Master Degree in Artificial Intelligence
A.Y. 2022/2023

Computer Vision & Deep Learning
Advanced Programming

Gaetano Alberto Caporusso - VR489760
Sebastiano D'Arconso - VR489066

Contents

1 GitHub

Here you'll find the GitHub page of the project Player-vs-Referee. The most important scripts are the *main.py* file and all the files in the *utils_proj* folder.

2 Motivation and rationale

Our project centers around creating an object detection system tailored specifically for NBA players and referees. By leveraging our self-created dataset, we aim to train a computer vision model capable of detecting and distinguish between these two specific classes. We wanted to combine our passion for NBA with the techincal skills acquired during the course and see if we were able to achieve this goal.

3 State of the Art

Over the past few years, YOLO has emerged as a groundbreaking and highly influential approach in the field of object detection. As of today, YOLOv7 is probably the benchmark for object detection algorithms, followed by EfficientDet, RetinaNet and Faster R-CNN.

4 Objectives

Our goal was to implement an object recognition model that could be able to detect and classify between two specific classes: NBA players and referees. We also wanted to try different models with different approaches in order to have different results to compare and see which model or approach could be the best for our specific task.

5 Methodology

In this section we will explain in detail the dataset selection and the models used for this project.

5.1 Dataset selection

Due to the unavailability of a pre-existing dataset containing NBA players and referees images, along with their respective bounding boxes, it was necessary for us to create our own dataset. This involved the process of sourcing relevant images on the internet and manually labelling each image with the corresponding bounding boxes. By undertaking this task, we ensured that we had a comprehensive and customized dataset specifically tailored to our

experiment's requirements. Each image of the dataset has been labeled using the *labelimg* tool.

5.1.1 Labels

We first labeled our images following the YOLO format, since YOLO was our first choice of model, then we used a python script to convert all the labels in COCO format, since we also wanted to try a Faster R-CNN trained on COCO.

```
# YOLO label format
<object-class> <x-center> <y-center> <width> <height>
```

For YOLO each image has to be associated to the corresponding labels.txt file, if there are more object in the same image each object class and bounding box informations are listed in the same labels.txt file. For COCO format the labels are a bit more complicated, since all the dataset has to be represented by a JSON file, the next structure represent a snippet of our JSON file and in particular the sections for the images and the labels (annotations):

```
# COCO label format
"images": [
     {
         "file_name": "test_10.jpg",
         "height": 256,
         "width": 256,
         "id": 0
     }
]
"annotations": [
     {
         "id": 0,
         "image_id": 0,
         "category_id": 2,   # class
         "bbox": [
             45,
             2,
             85,
             85
         ],
         "area": 7225,       # area of the bbox
         "segmentation": [], # empty because we don't provide a segmentation mask
         "iscrowd": 0        # object is not a crowd or group of instances
     }
]
```

Where under "images" there are all the basic informations about each image like its filename and dimensions, and under "annotations" there are

all the relevant informations of each image. In the COCO format the coordinates of the bounding box differ from the coordinates used in the YOLO format, in fact, in the COCO format the bounding box is represented by the x and y coordinates of the top-left corner and by the width and height of the box.

5.2 Dataset

(a) Referee train

(b) Player train

(c) Test image

Dataset	Number of images
Train	755
Test	157
Validation	54

5.3 Model selection

For our project we decided to try two different models:

- Faster R-CNN with two different backbones:
  - ResNet50
  - MobileNet
- YOLOv5s (s = small)

Model	Backbone	Classes	Params	Trained on
Faster R-CNN	MobileNet	91	19.4M	COCO
Faster R-CNN	ResNet50	91	38.2M	COCO
YOLOv5s	YOLOv5 v6.0	80	7.2M	COCO

6 Differences between the models

We decided to employ these two models for our task as they represent the two primary approaches, we were interested in exploring both methods to

gain a comprehensive understanding and evaluate their performance. Next we will describe the main differences between the two models, more in-depth differences will be discussed during the presentation.

6.1 Faster R-CNN

Faster R-CNN differs from its predecessors by a simple and yet powerful implementation, in fact, both R-CNN and Fast R-CNN use selective search to find the regions proposals. The selective search algorithm not only is slow and time-consuming but it's also a fixed algorithm, that means that no learning is involved during that process, and that can lead to bad proposals. Therefore, researchers came up with an object detection algorithm that eliminates the selective search and lets the network learn the region proposals. The initial pipeline is similar to the old versions: an image is provided as an input to a convolutional network which provides a convolutional feature map, but now, instead of using selective search on the feature map to identify the region proposals, a separate network is used to predict the region proposals. These regions proposals are then reshaped using a RoI pooling layer and then is used to classify the image within the proposed region. All these algorithms are region based algorithms.

6.2 YOLO

YOLO or You Only Look Once, is an object detection algorithm much different from the Faster R-CNN and similar, in fact, instead of using regions to localize objects within the image, YOLO splits the image in an SxS grid, and from that grid it takes m bounding boxes, then, for each bounding box the network outputs a class probability and offset values for the bounding box. So, in YOLO, a single convolutional network predicts the bounding boxes and the class probabilites for these boxes. The bounding box with a class probability above a threshold value is selected and used to locate the object within the image.

7 Experiments and results

In the next section, we will cover all the phases involved in the process, including training, testing, validation and the metrics used to evaluate the performances. All of the following are made on our own dataset.

7.1 Faster R-CNN - model preparation

In both cases we did transfer learning from pretrained networks and do the training process only on the last layers in order to achieve our goal. The pipeline is the following:

- Initialize the Faster R-CNN with the selected backbone.
- Retrieve the number of input features for the classifier score in the region of interest head of the model.
- Instanciate a *FastRCNNPredictor* class using the number of input features just retrieved and replacing the number of classes with our (3).
- Replace the box predictor of the roi heads with the new instance of the *FastRCNNPredictor* class.

In summary, we initialize a pre-trained Faster R-CNN model with ResNet50 or MobileNet backbone and FPN architecture, we retrieve the number of input features from the ROI heads, and replace the box predictor to customize the model's output for a specific number of classes, then we select the trainable parameters and we train only those parameters.

7.2 Faster R-CNN - dataset preparation

We had to split our dataset in three different subsets: train, test and validation. To do that we used a script that allowed us to select the percentage of the split. After that we used a python script to generate the *annotations.json* file for each subset, we then had to write a custom *Dataset* class in order to be able to load our own dataset. For the Faster R-CNN an *annotation.json* file has to be in every folder used as dataset.

7.2.1 The Dataset class

The implementation of the Dataset class follows the guide lines provided by PyTorch, so what we had to do was to implement three main functions: __init__, __len__, and __getitem__. The code can be seen on the GitHub page (link) and will be explained in-depth during the presentation but one feature that's worth explaining about the dataset class is that it returns the loaded image and the corresponding target. The image returned is a transformation of the original image of the dataset, the transformations applied to each image are the following:

```
def get_transforms(train=False):
    if train:
        transform = A.Compose([
            A.HorizontalFlip(p=0.3),
            A.VerticalFlip(p=0.3),
            A.RandomBrightnessContrast(p=0.1),
            A.ColorJitter(p=0.1),
            ToTensorV2()
        ], bbox_params=A.BboxParams(format='coco'))
    else:
        transform = A.Compose([
```

```
            ToTensorV2()
        ], bbox_params=A.BboxParams(format='coco'))

    return transform
```

For these transformations we used the *albumentations* library, which is useful beacuse it applies the same transformation on both the image and the corresponding bounding boxes, maintaining the correlation between the two. Since we are doing object classification we need to specify also the format of the bounding boxes. The target, on the other hand, is returned as follows:

```
## The target is first loaded from the annotations
def _load_target(self, id):
  return self.coco.loadAnns(self.coco.getAnnIds(id))

/.../

## Snippet at the end of the __getitem__ function
target = self._load_target(id)
targ = {} # here is our transformed target
targ['boxes'] = boxes
targ['labels'] = torch.tensor([t['category_id'] for t in target],
     dtype=torch.int64)
targ['image_id'] = torch.tensor([t['image_id'] for t in target])
targ['area'] = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0]) # we
     have a different area
targ['iscrowd'] = torch.tensor([t['iscrowd'] for t in target], dtype=torch.int64)
```

The last thing worth mentioning is that the boxes coordinates change from xywh to xyxy (xmin ymin xmax ymax). The dataset class is then used to load the three different subsets, and each subset is used in the *DataLoader* class of PyTorch to instantiate the different dataloaders.

7.3 Faster R-CNN - training

Now that we load correctly our data and we have modified the pretrained models in order to correctly use those data we can perform the actual training. For the Faster R-CNN, the main function that does the actual training is the *train_one_epoch* that iterates over the dataloader, takes the images and targets and gives them in input at our model set in train mode; it then prints some useful informations. The Faster R-CNN model provided by PyTorch automatically compute the losses if we pass the image and the target, so all we had to do was to store the output of the model (the losses) and perform the backpropagation. In particular, the model returns the losses as follows:

- **Loss classifier**: this is the loss associated to the classificiation task of the Faster R-CNN, after generating region proposal using the RPN, the

Faster R-CNN takes these proposals and classifies them into different object categories. The loss classifier measures the difference between the predicted class probabilities and the ground-truth class labels of the proposed regions.

- **Loss box regression**: the Faster R-CNN models also performs bounding box regression to refine the initially generated region proposals. The loss box regression measures the discrepancy between the predicted bounding box and the ground-truth bounding box coordinates for each proposed region.
- **Loss RPN box regression**: the RPN generates regions proposals by regressing bounding box coordinates and predicting their offsets from predefined anchor boxes. The loss RPN box regression calculates the difference between the predicted bounding box offset and the ground-truth offsets for the bounding boxes. This loss is specific to the RPN and helps refine the proposed regions.
- **Loss objectness**: the RPN is responsible for generating region proposal by classifying each anchor box as either an object or background. The loss objectness measures the difference between the predicted objectness scores and the ground-truth labels for the anchor boxes.

The actual training is done in the training function, which calls the one epoch training for a fixed number of iterations (training epochs).

7.4 Faster R-CNN - testing

For the test part, we use a pre-existing function called *evaluate*, this function can be found in the PyTorch/vision GitHub (Pytorch/vision) and its function is to evaluate and return statistics about the model, we'll talk about the results in the **results** chapter. The function *evaluate* is used as follows:

```
test_dataset = ds.Dataset(root=data_path, type=opt.mode,
    transforms=ds.get_transforms(True))
test_loader = DataLoader(dataset=test_dataset, batch_size=1, shuffle=False,
    collate_fn=ds.collate_fn)

## model loading
model_trained = torch.load(opt.weights, map_location=torch.device('cpu'))
model_trained.eval()

## evaluation of the model
evaluate(model_trained, test_loader, 'cpu')
```

As shown, the evaluate function takes the model trained, the dataloader of the test set and the device on which perform the evaluation.

7.5 Faster R-CNN - validation

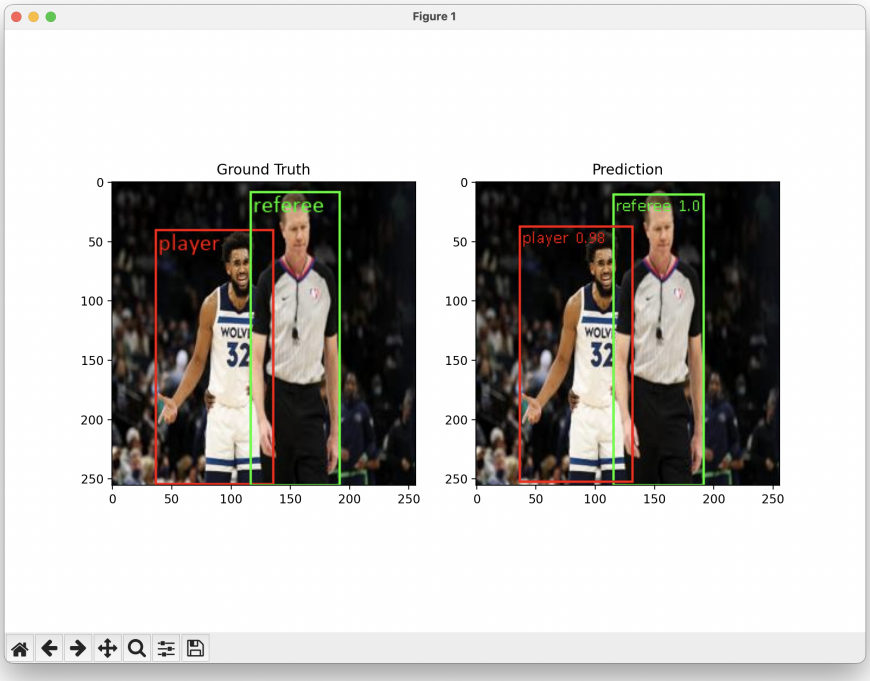
For the validation part we used an approach similar to the one used in the training phase, but this time we set the model in evaluation mode, we iterate over the validation set and we give every image of the set as an input to the model, we then plot both the ground truth and the output of the model. The output of the model in evaluation mode is much different from the output of the model in train mode. This time the output is:

```
# example of output
{'boxes': tensor[x, y, x, y], 'labels': tensor[2], 'scores': tensor[0.9915]}
```

where:

- **boxes**: this tensor represent the predict boundin boxes for the detected object(s) in the image.
- **labels**: is the tensor of predicted labels, for the detected object(s).
- **scores**: is the tensor of values of confidence scores for each bounding box.

During the validation process we decide to plot only the bounding boxes that have a score higher than a fixed threshold.
One output of the validation process:



7.6 Faster R-CNN - results

7.6.1 Training

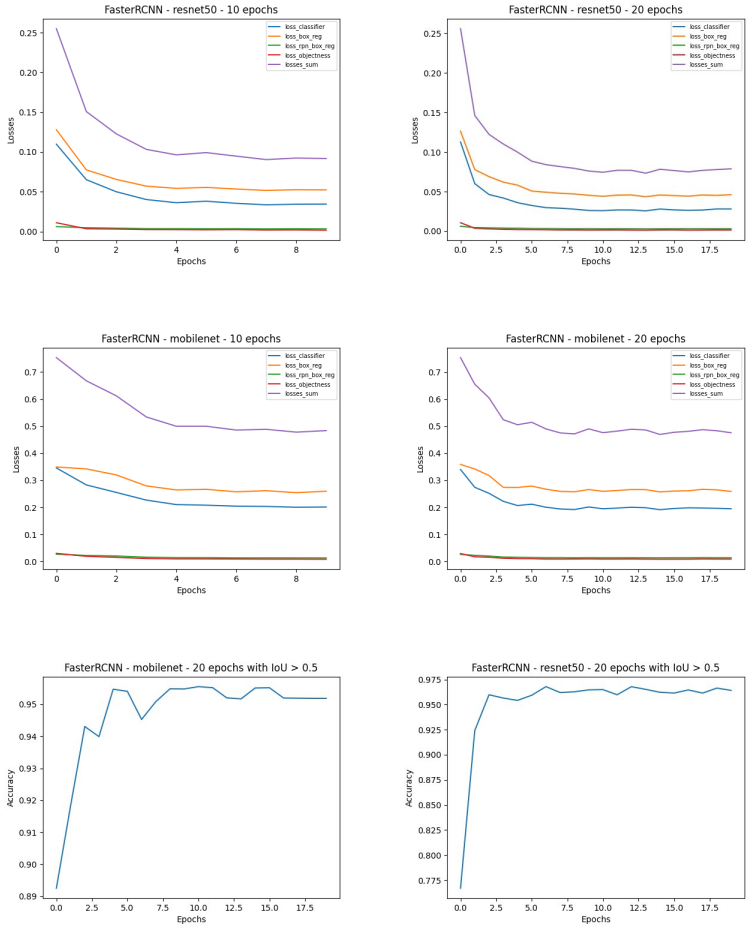


Figure 2: Losses and accuracy of IoU achieved during training

7.6.2 Testing

Statistics achieved during testing, using the *evaluate* function:

```
## ResNet50 10 epochs
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50   | area=  all | maxDets=100 ] = 0.942
```

```
## ResNet50 20 epochs
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50   | area=  all | maxDets=100 ] = 0.958
```

```
## MobileNet 10 epochs
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50   | area=  all | maxDets=100 ] = 0.951
```

```
## MobileNet 20 epochs
IoU metric: bbox
Average Precision (AP) @[ IoU=0.50   | area=  all | maxDets=100 ] = 0.959
```

7.7 YOLO - model preparation

For YOLO we followed the guide provided by the official ultralytics GitHub page (ultralytics), so we first cloned the yolov5 GitHub repo, we installed the requirements and downloaded the Yolov5s weights.

7.8 YOLO - dataset preparation

Since we already had the dataset with the labels in the YOLO format all we had to do was to split the dataset into the different subsets, then separate the images from the labels in each subset folder. Last but not least we had to do was to add a *dataset.yaml* file that is required for training the model.

```
train: /yolov5/data/images/train
val: /yolov5/data/images/test
# number of classes
nc: 2
# class names
names: ['player', 'referee']
```

7.9 YOLO - training

Running the train, once we are all set with the dataset we can actually run the train, using the command line:

```
python train.py --img 640 --batch 16 --epochs 20 --data dataset.yaml --weights
    yolov5s.pt
```

The training comprehends also the validation part and gives as output all the statistics of the both parts and also saves the last and the best weights of the model.

7.10 YOLO - inference

Since the train part comprehends also the validation, all we can do now is inference. This can be done by command line like before:

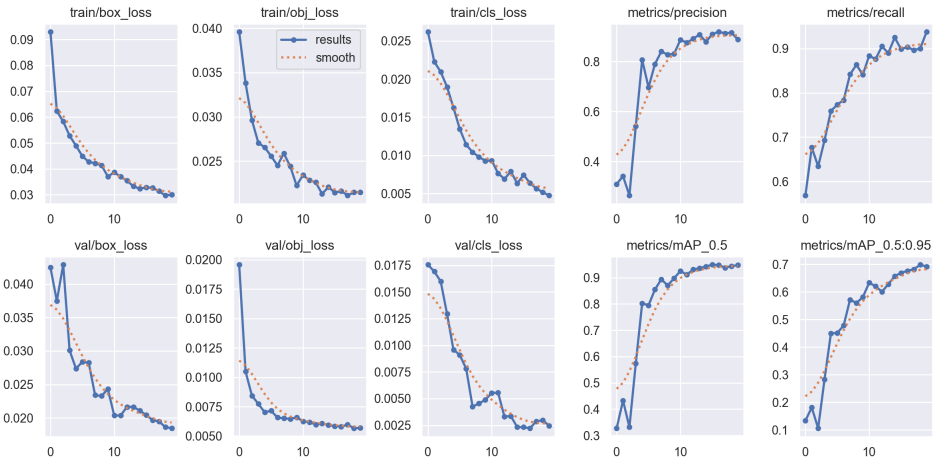
```
python detect.py --weights runs/train/exp/weights/best.pt --img 640 --conf 0.4
    --source image.jpg
```

or loading the weights of the model with pytorch and doing inference on it.

```
model = torch.hub.load('yolov5', 'custom', path='best.pt', source='local')
output = model(image)
```

7.11 YOLO - results

Results acquired during training and validation:



7.11.1 Tests

8 YOLO - ultralytics hub

Ultralytics offers a comprehensive service that enables you to streamline the process of training an user-chosen YOLO model using your dataset. It allows you to upload your dataset and to chose a model that you want to train on that dataset, it then opens a colab notebook and trains for 100 epochs. Once the training is complete, ultralytics provides you a range of statistics and evaluation metrics to access the performances of your model. With the ultralytics app, you also gain the ability to test your trained YOLO model in real time.

9 Advanced Programming

In this project, we also implemented some techinques seen in the Advanced Programming course, such as Few-Shot Learning and model compression methods, such as Pruning, Quantization and Knowledge Distillation.

9.1 Few-shot learning on Faster R-CNN with ResNet50

Few shot learning is a method that focuses on training to recognize and generalize patterns from a limited amount of data. The term "few-shot" refers to the fact that these models are designed to learn effectively with only a small number of examples per class during training. So, the initial requirement for conducting few-shot learning was to construct a reduced-size dataset. Note that the performance of the network on both datasets is categorized based on the optimizer used. We also decided to use, for all the experiments, a scheduler for the learning rate that multiplies the learing rate by 0.5 every 2 epochs.

9.1.1 Fewshot and smaller_fewshot datasets

For our purpose we decided to reduce our original dataset into two smaller datasets, **fewshot** dataset and **smaller_fewshot** dataset.

Name	Players	Referees	Both	Context	Total
fewshot	15	15	25	45	100
smaller_fewshot	11	10	17	32	70

The testing set is the same as before: 157 images.

(a) Player (b) Referee

(c) Player and referee (d) Context

Figure 4: Example of images in the fewshot and smaller_fewshot datasets

9.1.2 Experiments

All the experiments conducted were performed on a pretrained Faster R-CNN with backbone ResNet50, and the only variations made involved modifying hyperparameters to identify the optimal combination to achieve our objective. All the experiments differs from each other by:

- Freezed or not freezed backbone.
- Warm-up scheduler present or not.
- Train loader with batch of 1, 2, or 3 images.

Optimizers and scheduler:

```
## optimizers
optimizer_sgd = optim.SGD(params, lr=0.005, momentum = 0.9, weight_decay=0.0005)
optimizer_adam = optim.Adam(params, lr=0.005, weight_decay=0.0005)

## scheduler
lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.5) # for
    decrease the learning rate
warm_up_scheduler = optim.lr_scheduler.LambdaLR(optimizer, lambda epoch:
    min((epoch + 1) / 10, 1)) # for increase learning rate
```

Furthermore, in all experiments utilizing the warm-up scheduler, it increments the learning rate by 0.005 up to the fifth epoch, after which it is replaced by the standard lr_scheduler.

9.1.3 Performances on fewshot dataset

Note that the number of epochs to be used in the upcoming training is determined arbitrarily based on several tests conducted to identify the point at which the model's metrics no longer show improvement.

Optim	Epochs	Backbone	Warm-up	Batch	Precision
adam	15	freezed	yes	3	$\approx 90\%$
adam	15	not freezed	yes	3	$\approx 30\%$
adam	15	freezed	yes	1	$\approx 75\%$
adam	15	freezed	no	3	$> 90\%$
adam	15	not freezed	yes	1	$\approx 22\%$

Optim	Epochs	Backbone	Warm-up	Batch	Precision
SGD	15	freezed	yes	3	$\approx 80\%$
SGD	15	freezed	yes	1	$\approx 85\%$
SGD	15	not freezed	yes	3	$> 90\%$
SGD	15	not freezed	yes	1	$\approx 80\%$
SGD	15	not freezed	yes	3	$\approx 70\%$

9.1.4 Performances on smaller_fewshot dataset

Optim	Epochs	Backbone	Warm-up	Batch	Precision
adam	20	freezed	yes	2	$> 80\%$

Optim	Epochs	Backbone	Warm-up	Batch	Precision
SGD	20	not freezed	yes	2	$\approx 90\%$
SGD	20	not freezed	no	2	$\approx 60\%$

9.1.5 Few-shot learning - conclusions

Thanks to our experiments and the statistics acquired we came to these conclusions:

- fewshot dataset: with **Adam** the best combination is with **freezed backbone** and batch size of 3.
- fewshot dataset: with **SGD** the performances were really close but it exceeded when we used the **warm up scheduler**.
- smaller_fewshot dataset: with **Adam** the best combination is with **freezed backbone**, **warm up scheduler** and batch size of 2.

- smaller_fewshot dataset: with **SGD** overall we always achieved good performances but the best were when we used the **warm up scheduler**.

9.2 Pruning

Pruning is a technique used to optimize and compress neural networks by removing unnecessary connections or parameters. The primary goal of pruning is to reduce the model's size, memory footprint, and computational complexity while maintaining or even improving its performance. In a neural network, connections between neurons (weights) represent the parameters that are learned during the training process. However, not all connections contribute equally to the model's performance. Pruning identifies and removes the connections that have little impact on the overall accuracy of the model, these connections are, in fact, considered redundant or less informative. There are several pruning techniques, but one common approach is "weight pruning" or "magnitude pruning", which involves removing connections with small weights close to zero. This is usually done after the model has been trained and is based on the magnitude of the weights.

9.2.1 Objective

The objective was to prune our trained Faster R-CNN in order to increase its inference time. To do that we applied the weight pruning technique, then we did some tests on the pruned model in order to acquire informative statistics. We applied this method on both our trained models, the one with backbone ResNet50 and the one with backbone MobileNet.

```
modules_to_prune = []
## Pruning
for name, module in pruned_model.named_modules():
  if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
    modules_to_prune.append((name, module))
for name, module in modules_to_prune:
  prune.l1_unstructured(module, name='weight', amount=amount)
# Remove the pruning reparameterization buffers
for name, module in modules_to_prune:
  prune.remove(module, 'weight')
```

9.2.2 Tests

After we applied the pruning function to both our models (with backbone ResNet50 and with backbone MobileNet) we wanted to check if the changes occurred. Now we will show some results of the pruning method on the Faster R-CNN with MobileNet. Note that in this example we tried to prune the model by 33% (arbitrarily), total pruning time: **0.82s**.

Module	Sparsity
Full mobilenet	0.00%
Full pruned mobilenet	33.00%
Original backbone	0.00%
Pruned backbone	33.10%

We then inspected both the original and the pruned model in order to acquire some informations, we collected and compared statistics about **weights magnitudes**, **activations means** and **gradient magnitudes** in order to see if some changes occurred during the pruning procedure. We will display the first 5 layers of each model (ascending order).

Original model:

Layer	Weights	Activations	Gradient
Layer 0	0.0038	108.4763	0.0001
Layer 1	0.0048	113.3140	0.0001
Layer 2	0.0059	113.7022	0.0001
Layer 3	0.0068	116.6353	0.0001
Layer 4	0.0072	119.1553	0.0002

Pruned model:

Layer	Weights	Activations	Gradient
Layer 0	0.0035	108.6509	0.0001
Layer 1	0.0044	114.0956	0.0001
Layer 2	0.0055	114.2065	0.0001
Layer 3	0.0066	116.8959	0.0002
Layer 4	0.0067	118.1802	0.0002

And finally we did inference on both model on our validation to see if there were some changes in inference time:

Model	Time
Original model	15.47s
Pruned model	14.91s

9.3 Quantization

Quantization is a model compression technique that aims to reduce the precision or number of bits used to represent numerical values in a neural network. The primary goal of quantization is to compress the model's size and reduce its memory footprint and computational complexity while maintaining acceptable performance. In a neural network, weights, biases, and activations are typically represented using floating-point numbers, which can

have a high precision (e.g., 32 bits or 64 bits). However, using high precision numbers can be computationally expensive and memory-intensive, especially when deploying models on resource-constrained devices like mobile phones or embedded systems. Quantization involves converting the high precision floating-point numbers into lower precision fixed-point numbers or integers. For example, a common quantization technique is to use 8-bit integers to represent values instead of 32-bit floating-point numbers. This reduces the memory requirements and accelerates computations since fixed-point operations are often faster and more energy-efficient than floating-point operations on many hardware platforms. There are different types of quantization:

- Static quantization.
- Dynamic quantization.
- Static quantization aware training.

Since we intend to quantize a Faster R-CNN model, which is essentially a fusion of different sub-models, we need to quantize each individual part separately and then recombine them to form the complete quantized model. We decided to apply post training static quantization since is the only one that can quantize both convolutional and linear layers.

9.3.1 Tests

The quantization process consists in different parts, the first one separates the model in its sub-models:

- Backbone.
- Roi-heads.
- Region proposal network.

Then applies the quantization function to each of them:

```
## quantization function
def quantize_model(model):
  backend = 'qnnpack'
  model.qconfig = torch.quantization.get_default_qconfig(backend)
  torch.backends.quantized.engine = backend
  model_static_quantized = torch.quantization.prepare(model, inplace=True)
  model_static_quantized = torch.quantization.convert(model_static_quantized,
      inplace=True)
  return model_static_quantized
```

Then it fuses all the quantized modules back together and confronts the weights of both model files.

Name	Before quantization	After quantization
mobilenet	76.02MB	19.35MB
backbone	17.93MB	4.80MB
roi-heads	55.65MB	13.92MB
rpn	2.44MB	0.61MB

Name	Before quantization	After quantization
resnet50	173.40MB	44.01MB
backbone	107.74MB	27.55MB
roi-heads	60.91MB	15.26MB
rpn	4.74MB	1.19MB

9.3.2 Problems

We weren't able to perform inference on the quantized model due to the missing implementations of multiple quantized modules for the 'CPU' backend, we tried to do inference in different ways but still couldn't get to run the model.

9.4 Knowledge Distillation

To perform knowledge distillation, we initially designed a compact convolutional bacbone, which was then integrated with the Faster R-CNN framework as the student model.

```
## student model
def KD_model(n_classes):
  backbone = CustomBackbone()
  anchor_generator = AnchorGenerator(
     sizes=((32, 64, 128,256),), aspect_ratios=((0.5, 1.0, 2.0,2.5),))

  roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=['0', '1', '2',
       '3'],
                                                  output_size=7, sampling_ratio=2)
  model = FasterRCNN(backbone,n_classes,rpn_anchor_generator=anchor_generator,
       box_roi_pool=roi_pooler)

  return model
```

The teacher model, on the other hand, utilized the Faster R-CNN with the ResNet50 backbone, trained on our dataset. The training process followed a response-based approach, wherein we leveraged the response of the teacher model to guide and train the smaller student model effectively. During training, our main objective was to minimize various losses, including traditional ones and those related to the intersection over union (IoU). Altought we achieved promising results in minimizing these loss parameters,

the overall performance of the model in object detection tasks remained unsatisfactory. Despite our efforts in knowledge distillation and loss minimization, the student model's performance did not match our expectations in object detection. This outcome could be attributed to various factors, such as the complexity of the object detection task, the design limitations of the compact backbone, or the challenges in effectively transferring knowledge from the teacher to the student model. In conclusion, our attempts at knowledge distillation, utilizing a compact convolutional backbone and response-based training, showed potential in minimizing loss parameters. However, the model's object detection performance fell short of our desired results.

10 Conclusions

10.1 Computer Vision & Deep Learning

The project aimed to deploy and object detection model specifically designed for detecting NBA players and referees. Two popular object detection algorithms, Faster R-CNN and YOLO, were employed as the methods for this task. The Fast R-CNN algorithm is knowk for the precise bounding box prediction, while YOLO offers fast and efficient real-time object detection. The project involved several steps:

- Dataset collection and annotation.
- Model training.
- Model evaluation.

Despite having a relatively small dataset, the project managed to achieve decent results in NBA player and referee detection using the two models. This outcome highlights the inherent power and effectiveness of these models, even when trained on limited data. We are delighted with the results of our tests because despite the significant variations between the classes, the model performed well, even considering the considerable differences in the uniforms of players and the strikingly similar uniforms of referees. Therefore, the project's outcome not only demonstrates the effectiveness of these two models for object detection tasks but also emphasize the immense potential of these models when trained on a substantial amount of data.

10.2 Advandced Programming

In conclusion, our endeavors to implement various model compression and other advanced programming techinques demonstrated some level of effectiveness, even if not uniformely as we hoped. While we observed positive outcomes with few-shot learning, where the model exhibited an ability to

learn from limited data and generalize well, other techniques such as knowledge distillatio and quantization fell short of meeting our initial expectations. These methods showed some potential in specific aspects, such as minimizing loss parameters or lower the model weight, but they did not translate into substantial improvements in overall performance for our specific task of interest - object detection. On the other hand, pruning proved to be a valuable approach in reducing model size and inference time without compromising performance significantly. It demonstrated its usefulness particularly when applied to the Faster R-CNN with MobileNet architecture (probably due to its already optimized structure). In summary, while our exploration of model compression and advanced programming techniques revealed some successes, they did not uniformly deliver the desired outcomes. Few-shot learning and pruning stood out as the most promising strategies, highlighting the importance of carefully selecting and adapting techniques to specific tasks and architectures.