Published in Towards Data Science

Chris Hughes Follow

Nov 25, 2022 · 53 min read · Listen

YOLOv7: A Deep Dive into the Current State-of-the-Art for Object Detection

Everything you need to know to use YOLOv7 in custom training scripts

how this improved architecture surpasses all previous YOLO versions — as well as all other object detection models — in terms of both speed and accuracy on the MS COCO dataset; achieving this performance without utilizing any pretrained weights. Additionally, following all of the controversy around the naming conventions of previous YOLO models, as YOLOv7 was released by the same authors that developed Scaled-YOLOv4, the machine learning community seems happy to accept this as the next iteration of the 'official' YOLO family!

At the point that YOLOv7 was released, we — as part of Microsoft's Data & AI Service Line — were partway through a challenging object-detection based customer project, in a domain drastically different to COCO. Needless to say, both ourselves and the customer were very excited at the prospect of applying YOLOv7 to our problem. Unfortunately, when using the out-of-the-box settings, the results were … let's just say, not great.

After reading the official paper, we found that, whilst it presents a comprehensive overview on the architectural changes, it omits many details around how the model

was trained; for example, which data augmentation techniques were applied, and how the loss function measures that the model is doing a good job! To understand these technicalities, we decided to debug the code directly. However, as the YOLOv7 repository is a modified fork of th e — which itself is a fork of YOLOv5 — we found that it includes a lot of complex functionality, much of which is not needed when just training a model; for example, being able to specify custom architectures in Yaml format and have these translated into PyTorch models. Additionally, the codebase contains many custom components which have been implemented from scratch — such as a multi-GPU training loop, several data augmentations, samplers to preserve dataloader workers, and multiple learning rate schedulers — many of which are now available in PyTorch, or other libraries. As a result, there was a lot of code to dissect; it took us a long time to understand how everything worked, as well as the intricacies of the training loop which contribute to the model's excellent performance! Eventually, with this understanding, we were able to set up our training recipe to obtain consistently good results on our task.

In this article, we intend to take a practical approach in demonstrating how to train YOLOv7 models in custom training scripts, as well as exploring areas such as data augmentation techniques, how to select and modify anchor boxes, and demystifying how the loss function works; (hopefully!) enabling you to build up an intuition of what is likely to work well for your own problems. As the YOLOv7 architecture is well described in detail in the official paper, as well as in many other sources, we are not going to cover this here. Instead, we intend to focus on all of the other details which, whilst contribute to YOLOv7's performance, are **not** covered in the paper. This tends to be knowledge which has been accumulated over multiple versions of YOLO models but can be incredibly difficult to track down for someone just entering the field.

To illustrate these concepts, we shall be using our own implementation of YOLOv7, which utilises the official pretrained weights, but has been written with modularity and readability in mind. This project initially started as an exercise for us to improve our understanding of how YOLOv7 works under the hood — in order to better understand how to apply it — but after successfully using it on a few different tasks, we have decided to make it publicly available. Whilst we would recommend using the official implementation if you wish to exactly reproduce the published results on COCO, we find that this implementation is more flexible to apply, and

extend, to custom domains. Hopefully, this implementation will provide a clean and clear starting point for anyone wishing to experiment with YOLOv7 in their own custom training scripts, as well as providing more transparency around the techniques that were used during training in the original implementation.

In this article, we shall cover:

- Loading data in an appropriate format
- Using a pretrained model for inference
- Understanding the YOLOv7 Loss
- Finetuning on a new dataset
- Training from scratch

Exploring all of the details along the way, such as:

- Anchor boxes
- Feature Pyramid Networks (FPN)
- Center Priors
- Optimal Transport Assignment
- Mosaic Augmentation
- Mixup Augmentation
- Applying weight decay to parameter groups
- Cosine learning rate scheduling
- Gradient accumulation, and how this affects weight decay
- Model EMA
- Selecting appropriate anchor box sizes for your problem

Tl;dr: *If you just want to see some working code that you can use directly, all of the code required to replicate this post is available as a notebook here. Whilst code snippets are used*

throughout the article, this is primarily for aesthetic purposes, please defer to the notebook, and the repo for working code.

Acknowledgements

We would like to thank British Airways, for without their consistently delayed flights, this post would probably not have happened.

Data Loading

First, let's take a look at how to load our dataset in the format that YOLOv7 expects.

Selecting a dataset

Throughout this article, we shall use the Kaggle cars object detection dataset; however, as our aim is to demonstrate how YOLOv7 can be applied to any problem, this is really the least important part of this work. Additionally, as the images are quite similar to COCO, it will enable us to experiment with a pretrained model before we do any training.

The annotations for this dataset are in the form of a .csv file, which associates the image name with the corresponding annotations; where each row represents one bounding box. Whilst there are around 1000 images in the training set, only those with annotations are included in this file.

We can view the format of this by loading it into a pandas DataFrame.

```
pd.read_csv(annotations_file_path)
```

	image	xmin	ymin	xmax	ymax
0	vid_4_1000.jpg	281.259045	187.035071	327.727931	223.225547
1	vid_4_10000.jpg	15.163531	187.035071	120.329957	236.430180
2	vid_4_10040.jpg	239.192475	176.764801	361.968162	236.430180
3	vid_4_10020.jpg	496.483358	172.363256	630.020260	231.539575
4	vid_4_10060.jpg	16.630970	186.546010	132.558611	238.386422
...	...	...	...	...	...
554	vid_4_9860.jpg	0.000000	198.321729	49.235251	236.223284
555	vid_4_9880.jpg	329.876184	156.482351	536.664239	250.497895
556	vid_4_9900.jpg	0.000000	168.295823	141.797524	239.176652
557	vid_4_9960.jpg	487.428988	172.233646	616.917699	228.839864
558	vid_4_9980.jpg	221.558631	182.570434	348.585579	238.192196

559 rows × 5 columns

As it is not usually the case that all images in our dataset contain instances of the objects that we are trying to detect, we would also like to include some images that do not contain cars. To do this, we can define a function to load the annotations which also includes 100 ‘negative’ images. Additionally, as the designated test set is unlabelled, let’s randomly take 20% of these images to use as our validation set.

```
    annotations_df = pd.read_csv(annotations_file_path)
    annotations_df.loc[:, "class_name"] = "car"
    annotations_df.loc[:, "has_annotation"] = True

    # add 100 empty images to the dataset
    empty_images = sorted(set(all_images) - set(annotations_df.image.unique()))
    non_annotated_df = pd.DataFrame(list(empty_images)[:100], columns=["image"])
    non_annotated_df.loc[:, "has_annotation"] = False
    non_annotated_df.loc[:, "class_name"] = "background"

    df = pd.concat((annotations_df, non_annotated_df))

    class_id_to_label = dict(
        enumerate(df.query("has_annotation == True").class_name.unique())
    )
    class_label_to_id = {v: k for k, v in class_id_to_label.items()}

    df["image_id"] = df.image.map(image_to_image_id)
    df["class_id"] = df.class_name.map(class_label_to_id)

    file_names = tuple(df.image.unique())
    random.seed(42)
    validation_files = set(random.sample(file_names, int(len(df) * 0.2)))
    train_df = df[~df.image.isin(validation_files)]
    valid_df = df[df.image.isin(validation_files)]

    lookups = {
        "image_id_to_image": image_id_to_image,
        "image_to_image_id": image_to_image_id,
        "class_id_to_label": class_id_to_label,
        "class_label_to_id": class_label_to_id,
    }
    return train_df, valid_df, lookups
```

We can now use this function to load our data:

```
train_df, valid_df, lookups = load_cars_df(annotations_file_path, images_path)
```

```
train_df
```

	image	xmin	ymin	xmax	ymax	class_name	has_annotation	image_id	class_id
0	vid_4_1000.jpg	281.259045	187.035071	327.727931	223.225547	car	True	0	0.0
1	vid_4_10000.jpg	15.163531	187.035071	120.329957	236.430180	car	True	1	0.0
2	vid_4_10040.jpg	239.192475	176.764801	361.968162	236.430180	car	True	3	0.0
4	vid_4_10060.jpg	16.630970	186.546010	132.558611	238.386422	car	True	4	0.0
5	vid_4_10100.jpg	447.568741	160.625804	582.083936	232.517696	car	True	6	0.0
...	...	...	...	...	...	...	...	...	...
91	vid_4_13060.jpg	NaN	NaN	NaN	NaN	background	False	142	NaN
93	vid_4_13100.jpg	NaN	NaN	NaN	NaN	background	False	144	NaN
95	vid_4_13240.jpg	NaN	NaN	NaN	NaN	background	False	146	NaN
97	vid_4_13280.jpg	NaN	NaN	NaN	NaN	background	False	148	NaN
98	vid_4_13300.jpg	NaN	NaN	NaN	NaN	background	False	149	NaN

465 rows × 9 columns

To make it easier to associate predictions with an image, we have assigned each image a unique id; in this case it is just an incrementing integer count. Additionally, we have added an integer value to represent the classes that we want to detect, which is a single class — ‘car’ — in this case.

Generally, object detection models reserve `0` as the background class, so class labels should start from `1`. This is **not** the case for YOLOv7, so we start our class encoding from `0`. For images that do not contain a car, we do not require a class id. We can confirm that this is the case by inspecting the lookups returned by our function.

```
lookups.keys()
```

```
dict_keys(['image_id_to_image', 'image_to_image_id', 'class_id_to_label', 'class_label_to_id'])
```

```
lookups['class_label_to_id'], lookups['class_id_to_label']
```

```
({'car': 0}, {0: 'car'})
```

Finally, let’s see the number of images in each class for our training and validation sets. As an image can have multiple annotations, we need to make sure that we account for this when calculating our counts:

```
print(f"Num. annotated images in training set: {len(train_df.query('has_annotation == True').image.unique())}")
print(f"Num. Background images in training set: {len(train_df.query('has_annotation == False').image.unique())}")
print(f"Total Num. images in training set: {len(train_df.image.unique())}")
print('------------')

print(f"Num. annotated images in validation set: {len(valid_df.query('has_annotation == True').image.unique())}")
print(f"Num. Background images in validation set: {len(valid_df.query('has_annotation == False').image.unique())}")
print(f"Total Num. images in validation set: {len(valid_df.image.unique())}")
```

```
Num. annotated images in training set: 256
Num. Background images in training set: 68
Total Num. images in training set: 324
------------
Num. annotated images in validation set: 99
Num. Background images in validation set: 32
Total Num. images in validation set: 131
```

Create a Dataset Adaptor

Usually, at this point, we would create a PyTorch dataset specific to the model that we shall be training.

However, we often use the pattern of first creating a dataset 'adaptor' class, with the sole responsibility of wrapping the underlying data sources and loading this appropriately. This way, we can easily switch out adaptors when using different datasets, without changing any pre-processing logic which is specific to the model that we are training.

Therefore, let's focus for now on creating a `CarsDatasetAdaptor` class, which converts the specific raw dataset format into an image and corresponding annotations. Additionally, let's load the image id that we assigned, as well as the height and width of our image, as they may be useful to us later on.

An implementation of this is presented below:

```
            self.image_idx_to_image_id = {
                idx: image_id
                for idx, image_id in enumerate(self.annotations_df.image_id.unique(
            }
            self.image_id_to_image_idx = {
                v: k for k, v, in self.image_idx_to_image_id.items()
            }

        def __len__(self) -> int:
            return len(self.image_idx_to_image_id)

        def __getitem__(self, index):
            image_id = self.image_idx_to_image_id[index]
            image_info = self.annotations_df[self.annotations_df.image_id == image_
            file_name = image_info.image.values[0]
            assert image_id == image_info.image_id.values[0]

            image = Image.open(self.images_dir_path / file_name).convert("RGB")
            image = np.array(image)

            image_hw = image.shape[:2]

            if image_info.has_annotation.any():
                xyxy_bboxes = image_info[["xmin", "ymin", "xmax", "ymax"]].values
                class_ids = image_info["class_id"].values
            else:
                xyxy_bboxes = np.array([])
                class_ids = np.array([])

            if self.transforms is not None:
                transformed = self.transforms(
                    image=image, bboxes=xyxy_bboxes, labels=class_ids
                )
                image = transformed["image"]
                xyxy_bboxes = np.array(transformed["bboxes"])
                class_ids = np.array(transformed["labels"])

            return image, xyxy_bboxes, class_ids, image_id, image_hw
```

Notice that, for our background images, we are just returning an empty array for our bounding boxes and class ids.

Using this, we can confirm that the length of our dataset is the same as the total number of training images that we calculated earlier.

```
ds = CarsDatasetAdaptor(images_path, train_df)
```

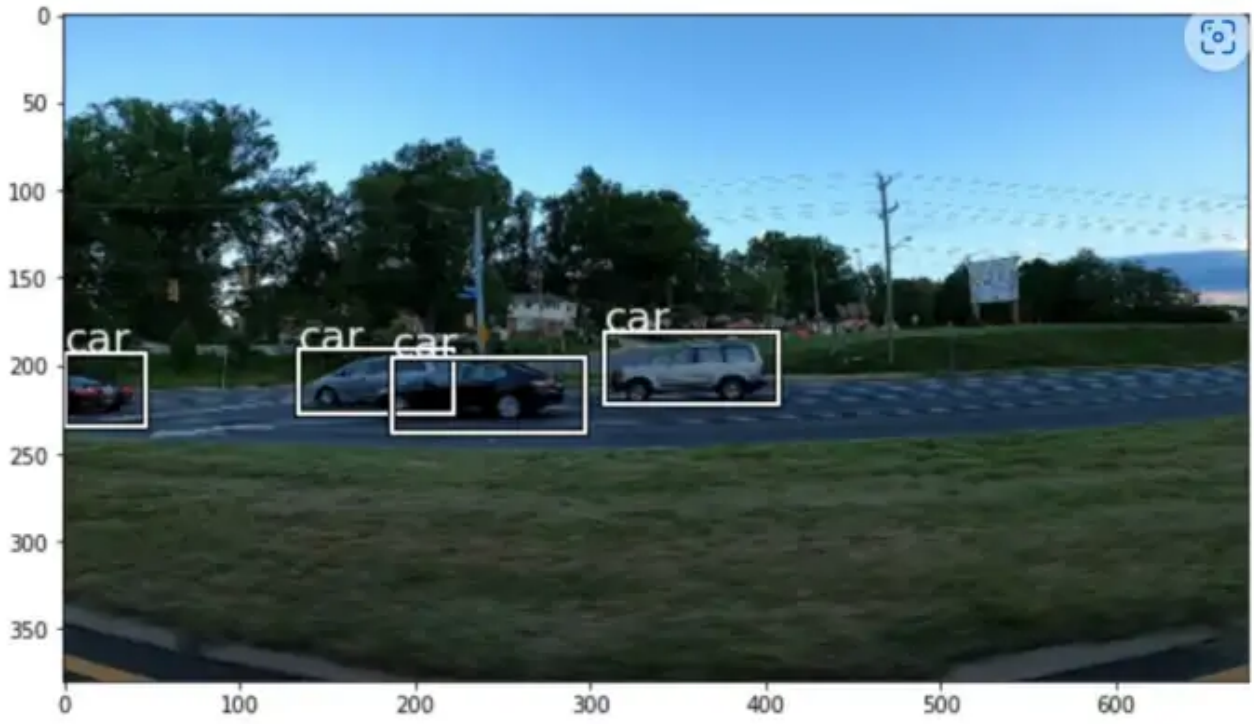
```
len(ds)
```

```
324
```

Now, we can use this to visualise some of our images, as demonstrated below:

```
idx = 4
image, xyxy_bboxes, class_ids, image_idx, image_size = ds[idx]
show_image(image, xyxy_bboxes.tolist(), [lookups['class_id_to_label'][int(c)] for c in class_ids])
print(f'Image id: {image_idx}')
print(f'Image size: {image_size}')
```


```
Image id: 6
Image size: (380, 676)
```




```
Image id: 848
Image size: (380, 676)
```

Create a YOLOv7 dataset

Now that we have created our dataset adaptor, let's create a dataset which preprocesses our inputs into the format required by YOLOv7; these steps should remain the same regardless of the adaptor that we are using.

An implementation of this is presented below:

Let's wrap our data adaptor using this dataset and inspect some of the outputs:

```
from yolov7.dataset import Yolov7Dataset
```

```
yolo_ds = Yolov7Dataset(ds)
```

```
idx = 4
image_tensor, labels, image_id, image_size = yolo_ds[idx]

print(f'Image: {image_tensor.shape}')
print(f'Labels: {labels}')

# denormalize boxes
boxes = labels[:, 2:]
boxes[:, [0, 2]] *= image_size[1]
boxes[:, [1, 3]] *= image_size[0]

show_image(image_tensor.permute( 1, 2, 0), boxes.tolist(), [lookups['class_id_to_label'][int(c)] for c in labels[:, 1]], 'cxcywh')
print(f'Image id: {image_id}')
print(f'Image size: {image_size}')
```

```
Image: torch.Size([3, 380, 676])
Labels: tensor([[0.0000, 0.0000, 0.7616, 0.5173, 0.1990, 0.1892]])
```


```
Image id: 6
Image size: tensor([380, 676])
```

As we haven't defined any transforms, the output is largely the same, with the main exception being that the boxes are now in normalized cxcywh format and all of our outputs have been converted into tensors. Note that `cx, cy` stands for center x and y and it means that the coordinates correspond to the centre of the box.

One thing to note is that our labels take the form `[0, class_id, ncx, ncy, nw, nh]`. The zero space at the start of the tensor will be utilised by the collate function later on.

Transforms

Now, let's define some transforms! For this, we shall use the excellent Albumentations library, which provides many options for transforming both images

and bounding boxes.

Whilst the transforms that we select will largely be domain specific, here, we shall ‘s define similar transforms to those used in the original implementation.

These are:

- Resize the image to the given input (multiple of 640) whilst maintaining the aspect ratio
- If the image is not square, apply padding. For this, we shall follow the paper in using a grey padding, this is an arbitrary choice.

During training:

- Horizontal flip.

We can use the following function to create these transforms as demonstrated below:

```
        bbox_params=A.BboxParams(format="pascal_voc", label_fields=["labels"]),
    )
```

Now, let's re-create our dataset, this time passing the default transforms that will be used during evaluation. For our target image size, we shall use `640` which is the value that the smaller YOLOv7 models were trained on. In general, we can select any multiple of 8 for this.

```
from yolov7.dataset import create_yolov7_transforms
```

```
target_image_size = 640
```

```
yolo_ds = Yolov7Dataset(ds, transforms=create_yolov7_transforms(image_size=(target_image_size, target_image_size)))
```

```
idx = 4
image_tensor, labels, image_id, image_size = yolo_ds[idx]

boxes = labels[:, 2:]
boxes[:, [0, 2]] *= target_image_size
boxes[:, [1, 3]] *= target_image_size

show_image(image_tensor.permute( 1, 2, 0), boxes.tolist(), [lookups['class_id_to_label'][int(c)] for c in labels[:, 1]], 'cxcywh')
print(f'Image id: {image_id}')
print(f'Original Image size: {image_size}')
print(f'Resized Image size: {image_tensor.shape[1:]}')
```


```
Image id: 6
Original Image size: tensor([380, 676])
Resized Image size: torch.Size([640, 640])
```

Using these transforms, we can see that our image has been resized to our target size and padding has been applied. The reason that padding is used is so that we can maintain the aspect ratio of the objects in the images, but have a common size for images in our dataset; enabling us to batch them efficiently!

Using a pretrained model

Now that we have explored how to load and prepare our data, let's move on to take a look at how we can leverage a pretrained model to make some predictions!

Loading the model

So that we can understand how to interface with the model, let's load a pretrained checkpoint and use this for inference on some images in our dataset. As this checkpoint was trained on COCO, which contains images of cars, we can assume that the model should perform moderately well on this task out of the box. To see the models that are available, we can import the `AVAILABLE_MODELS` variable.

```
from yolov7 import AVAILABLE_MODELS

AVAILABLE_MODELS
```

```
['yolov7',
 'yolov7x',
 'yolov7-tiny',
 'yolov7-w6',
 'yolov7-d6',
 'yolov7-e6',
 'yolov7-e6e']
```

Here, we can see that the available models are the architectures defined in the original paper. Let's create the standard `yolov7` model, using the `create_yolov7_model` function.

```
from yolov7 import create_yolov7_model
```

```
model = create_yolov7_model('yolov7', num_classes=80, pretrained=True)
model.eval();
```

```
Transferred 564/566 items from https://github.com/Chris-hughes10/Yolov7-training/releases/download/0.1.0/yolov7_training_state_dict.pt
```

Now, let's take a look at the model's predictions. The forward pass through the model will return the raw feature maps given by the FPN heads, to convert these into meaningful predictions, we can use the `postprocess` method.

```
import torch

with torch.no_grad():
    model_outputs = model(image_tensor[None])
    preds = model.postprocess(model_outputs, conf_thres=0., multiple_labels_per_box=False)
```

```
preds[0].shape
```

```
torch.Size([25200, 6])
```

Inspecting the shape, we can see that the model has made 25,200 predictions! Each prediction has an associated tensor of length 6 — the entries correspond to the bounding box coordinates in xyxy format, a confidence score, and a class index.

Often, object detection models tend to make a lot of similar, overlapping predictions. Whilst there are many ways of dealing with this, in the original paper, the authors used non-maximum-suppression (NMS) to solve this problem. We can apply NMS, as well as a secondary round of confidence thresholding, using the function below. In addition, during postprocessing, we often want to filter our any predictions with a confidence level below a predefined threshold, let's increase our confidence threshold here.

```
                boxes=pred[:, :4],
                scores=pred[:, 4],
                idxs=pred[:, 5],
                iou_threshold=nms_threshold,
            )
            nms_preds.append(pred[nms_idx])

        return nms_preds
```

```
from yolov7.trainer import filter_eval_predictions
```

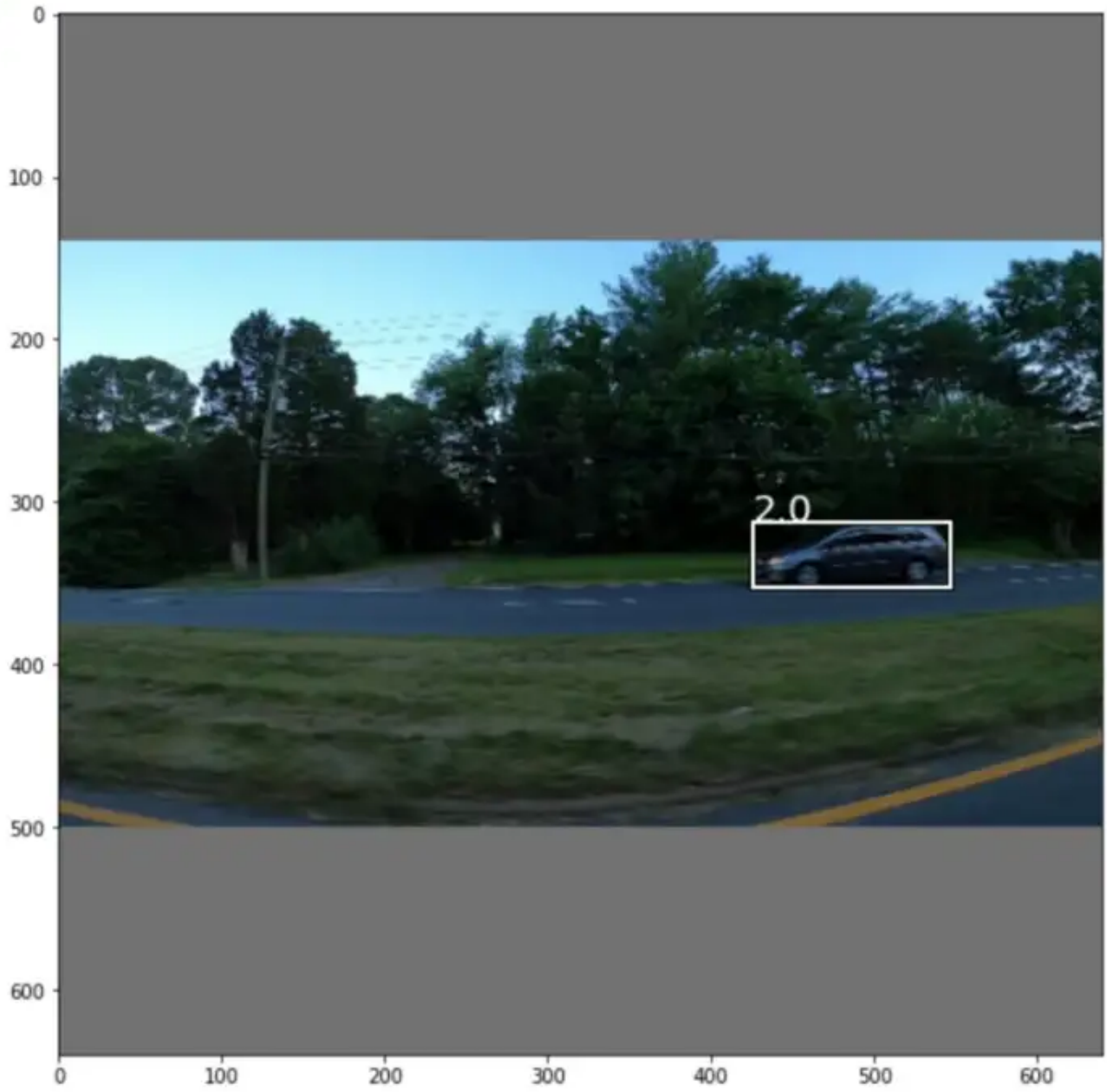
```
nms_predictions = filter_eval_predictions(preds, confidence_threshold=0.1)
```

```
nms_predictions[0].shape
```

```
torch.Size([1, 6])
```

After applying NMS, we can see that now we only have a single prediction for this image. Let's visualise how this looks:

```
boxes = nms_predictions[0][:, :4]
class_ids = nms_predictions[0][:, -1]

show_image(image_tensor.permute( 1, 2, 0), boxes.tolist(), class_ids.tolist())
print(f'Image id: {image_id}')
print(f'Original Image size: {image_size}')
print(f'Resized Image size: {image_tensor.shape[1:]}')
```




```
Image id: 6
Original Image size: tensor([380, 676])
Resized Image size: torch.Size([640, 640])
```

We can see that this looks pretty good! The prediction from the model is actually tighter around the car than the ground truth!

Now that we have our prediction, the only thing to note is that the bounding box is relative to the *resized* image size. To scale our predictions back to the original image size, we can use the following function:

```
# https://github.com/Chris-hughes10/Yolov7-training/blob/main/yolov7/trainer.py

def scale_bboxes_to_original_image_size(
    xyxy_boxes, resized_hw, original_hw, is_padded=True
):
    scaled_boxes = xyxy_boxes.clone()
    scale_ratio = resized_hw[0] / original_hw[0], resized_hw[1] / original_hw[1

    if is_padded:
        # remove padding
        pad_scale = min(scale_ratio)
        padding = (resized_hw[1] - original_hw[1] * pad_scale) / 2, (
            resized_hw[0] - original_hw[0] * pad_scale
        ) / 2
        scaled_boxes[:, [0, 2]] -= padding[0]  # x padding
        scaled_boxes[:, [1, 3]] -= padding[1]  # y padding
        scale_ratio = (pad_scale, pad_scale)

    scaled_boxes[:, [0, 2]] /= scale_ratio[1]
    scaled_boxes[:, [1, 3]] /= scale_ratio[0]

    # Clip bounding xyxy bounding boxes to image shape (height, width)
    scaled_boxes[:, 0].clamp_(0, original_hw[1])  # x1
    scaled_boxes[:, 1].clamp_(0, original_hw[0])  # y1
    scaled_boxes[:, 2].clamp_(0, original_hw[1])  # x2
    scaled_boxes[:, 3].clamp_(0, original_hw[0])  # y2

    return scaled_boxes
```

```
from yolov7.models.yolo import scale_bboxes_to_original_image_size
```

```
idx = 4
image, _, _, image_idx, original_image_size = ds[idx]
show_image(image,
           scale_bboxes_to_original_image_size(boxes, original_hw=original_image_size, resized_hw=image_tensor.shape[1:]).tolist(),
           class_ids.tolist())
print(f'Image id: {image_idx}')
print(f'Image size: {image_size}')
```


```
Image id: 6
Image size: tensor([380, 676])
```

Understanding the Loss

Before we can start training, in addition to a model architecture, we need a loss function which will enable us to measure how well our model is performing; in order to be able to update our parameters. Since Object Detection is a difficult problem to teach a model, the loss functions of such models are usually quite complex and YOLOv7 is not an exception. Here, we shall do our best to illustrate the intuitions behind it to facilitate its understanding.

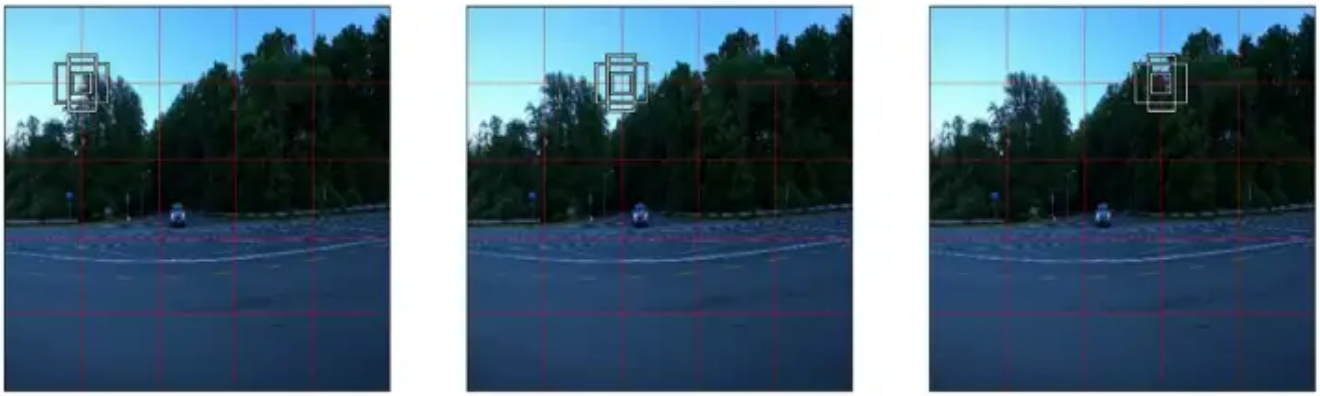
Before we can delve deeper into the actual loss function, let's cover a few background concepts that we need to understand.

Anchor boxes

One of the main difficulties of object detection is outputting detection boxes. That is, how do we train a model to create a bounding box and localize it correctly in an image?

There are a few different approaches, but the YOLOv7 family is what we call an **anchor-based** model. In these models, the general philosophy is to first create lots of potential bounding boxes, then select the most promising options to match to our target objects; slightly moving and resizing them as necessary to obtain the best possible fit.

The basic idea is that we draw a grid on top of each image and, at each grid intersection (**anchor point**), generate candidate boxes (**anchor boxes**) based on a number of **anchor sizes**. That is, the same set of boxes is repeated at each anchor point. This way, the task that model has to learn, slightly relocating and resizing these boxes, is simpler than generating boxes from scratch.



An example of anchor boxes generated at a sample of anchor points.

However, one issue with this approach is that our target, ground truth, boxes can range in size — from tiny to huge! Therefore, it is usually not possible to define a single set of anchor sizes that can be matched to all targets. For this reason, anchor-based model architectures usually employ a Feature-Pyramid-Network (FPN) to assist with this; which is the case with YOLOv7.

Feature Pyramid Networks (FPN)

The main idea behind FPNs (introduced in Feature Pyramid Networks for Object Detection) is to leverage the nature of convolutional layers — which reduce the size of the feature space and increase the coverage of each feature in the initial image — to output predictions at different scales[1]. FPNs are usually implemented as a stack of convolutional layers, as we can see by inspecting the detection head of our YOLOv7 model.

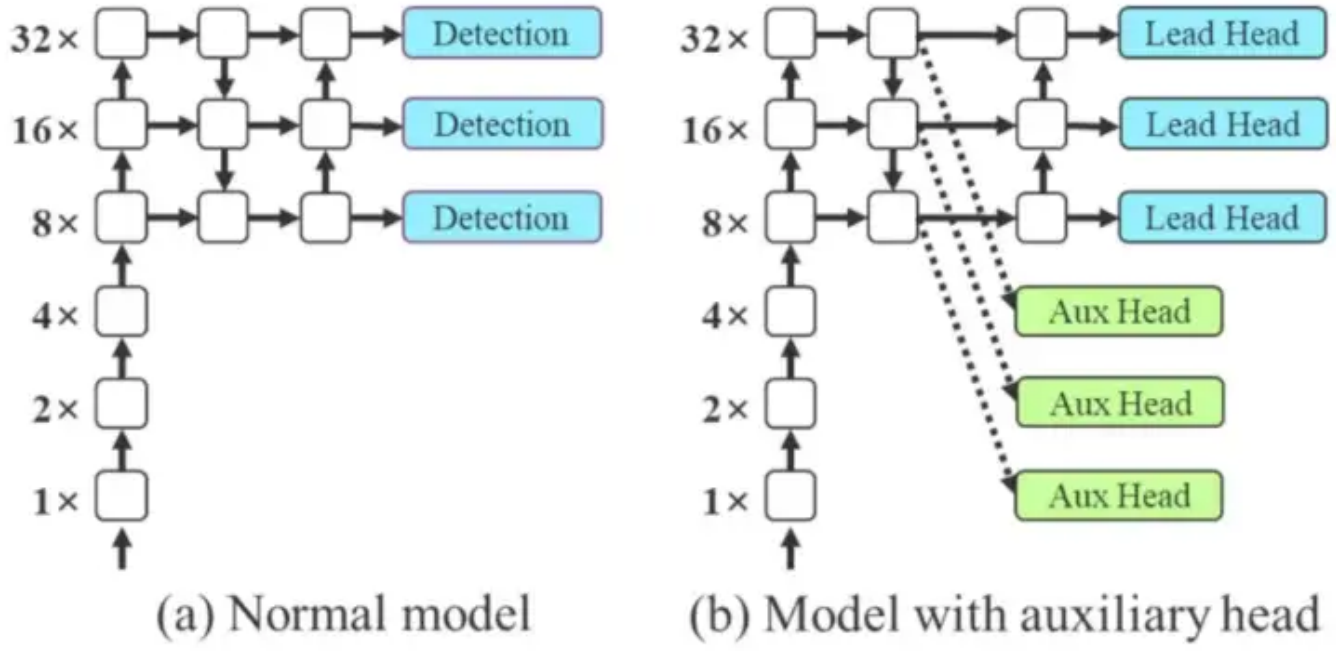
```
model.detection_head
```

```
Yolov7DetectionHead(
  (m): ModuleList(
    (0): Conv2d(256, 255, kernel_size=(1, 1), stride=(1, 1))
    (1): Conv2d(512, 255, kernel_size=(1, 1), stride=(1, 1))
    (2): Conv2d(1024, 255, kernel_size=(1, 1), stride=(1, 1))
  )
  (ia): ModuleList(
    (0): ImplicitAdd()
    (1): ImplicitAdd()
    (2): ImplicitAdd()
  )
  (im): ModuleList(
    (0): ImplicitMultiply()
    (1): ImplicitMultiply()
    (2): ImplicitMultiply()
  )
)
```

Whilst we could simply take the outputs of the final layer as predictions, as the deeper convolutional layers implicitly utilise the information from previous layers to learn more high-level features, they do not have access to the information of how to detect the lower-level features contained in earlier layers; this can result in poor performance when detecting smaller objects.

For this reason, a top-down pathway and lateral connections are added to the regular bottom-up pathway (normal flow of a convolution layer). The top-down pathway hallucinates higher resolution features by upsampling spatially coarser, but semantically stronger, feature maps from higher pyramid levels. Then, these features are enhanced with features from the bottom-up pathway through the lateral connections. The bottom-up feature map is of lower-level semantics, but its activations are more accurately localized as it was subsampled fewer times[1].

In summary, FPNs provide semantically strong features at multiple scales which make them extremely well suited for object detection. The connections that YOLOv7 implements in its FPN are illustrated in the figure below:



Representation of the YOLOv7 family Feature Proposal Network architecture. Source: YOLOv7 paper.

Here, we can see that we have a "Normal model" and a "Model with auxiliary head". This is because some of the larger models in the YOLOv7 family use deep supervision when training; that is, they leverage the outputs of deeper layers in the loss in order to try and better learn the task. We shall explore this further later on.

From the image, we can see that each layer in the FPN (also known as each FPN head), has a feature scale that is half the size of the previous one (the scale is the same for each Lead head and its corresponding Aux head). This can be understood as each subsequent FPN head "seeing" object scales twice as big as the previous one. We can leverage that by assigning grids with different **strides** (grid cell side size), and proportional anchor sizes, to each FPN head.

For instance, the anchor configuration for the basic `yolov7` model looks like this:

Illustration of the anchor grid and the different (default) anchor box sizes for each fpn head in the main model in the YOLOv7 family

As we can see, we have anchor box sizes and grids that cover completely different scales: from tiny objects to objects that can occupy the whole image.

Now, that we understand these ideas conceptually, let's take a look at the FPN outputs that come out of our model, which is what will be used to calculate our loss.

[1] *These sections were directly taken from the original FPN paper as we felt that no further explanation was needed.*

Breaking down the FPN outputs

Recall that, when we made our predictions earlier, we used the model's `postprocess` method to convert the raw FPN outputs into usable bounding boxes. Now that we understand the intuition behind what the FPN is trying to do, let's inspect these raw outputs.

```
with torch.no_grad():
    fpn_outputs = model(image_tensor[None])
```

The outputs of our model are always a `List[Tensor]`, where each component corresponds to a head of the FPN. For models that use Deep Supervision, the Aux Head outputs come after the Lead Head outputs (always the same number of each, both sides of the pair ordered equally). For the rest, including the one we are using here, only the Lead Head outputs are present.

```
type(fpn_outputs), len(fpn_outputs)
```

```
(list, 3)
```

```
for output in fpn_outputs:
    print(output.shape)
```

```
torch.Size([1, 3, 80, 80, 85])
torch.Size([1, 3, 40, 40, 85])
torch.Size([1, 3, 20, 20, 85])
```

Inspecting the shape of each FPN output, we can see that each one has the following dimensions:

```
[n_images, n_anchor_sizes, n_grid_rows, n_grid_cols, n_features]
```

where:

- `n_images` — The number of images in the batch (batch size).
- `n_anchor_sizes` - The anchor sizes associated with the head (usually 3).
- `n_grid_rows` - The number of anchors vertically, `img_height / stride`.
- `n_grid_cols` - The number of anchors horizontally, `img_width / stride`.
- `n_features - 5 + num_classes`
  - `cx` - Horizontal correction for the anchor box centre.
  - `cy` - Vertical correction for the anchor box centre.
  - `w` - Width correction for the anchor box.
  - `h` - Height correction for the anchor box.
  - `obj_score` - Score proportional to the probability of an object being contained inside the anchor box.

- `cls_score` - One per class, score proportional to the probability of that being the class of the object.

When these outputs are mapped into useful predictions during post-processing, we apply the following operations:

- `cx, cy : final = 2 * sigmoid(initial) - 0.5`
  [(−∞, ∞), (−∞, ∞)] → [(−0.5, 1.5), (−0.5, 1.5)]
  - The model can only move the anchor centre from 0.5 cell behind to 1.5 cells forward. Note that for the loss (i.e., when we train) we use grid coordinates.

- `w, h : final = (2 * sigmoid(initial)**2`
  [(−∞, ∞), (−∞, ∞)] → [(0, 4), (0, 4)]
  - The model can make the anchor box arbitrarily smaller but at most 4 times bigger. Larger objects, outside of this range, must be predicted by the next FPN head.

- `obj_score : final = sigmoid(initial)`
  (−∞, ∞) → (0, 1)
  - Makes sure the score is mapped to a probability.

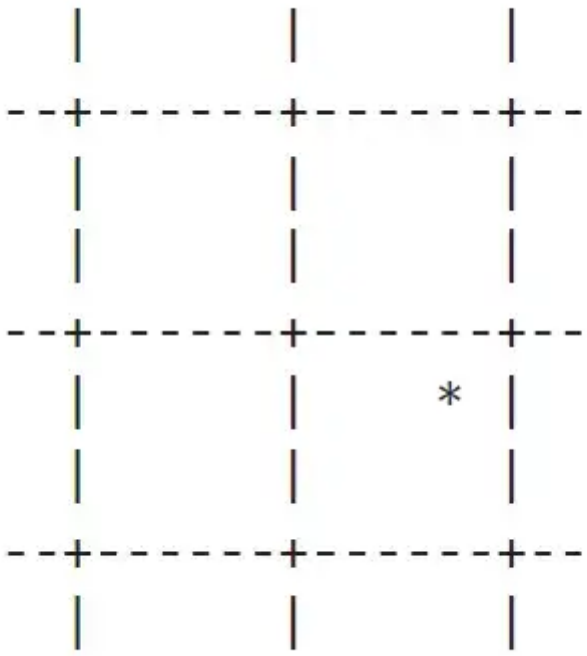
- `cls_score : final = sigmoid(initial)`
  (−∞, ∞) → (0, 1)
  - Makes sure the score is mapped to a probability.

Center Priors

Now, it is easy to see that if we put 3 anchor boxes in each anchor point of each of the grids, we end up with a lot of boxes: `3*80*80 + 3*40*40 + 3*20*20=25200` for each 640x640px image to be exact! The issue is that most of these predictions are not going to contain an object, which we classify as 'background'. Depending on the sequence of operations that we need to apply to each prediction, computations can easily stack up and slow down the training!

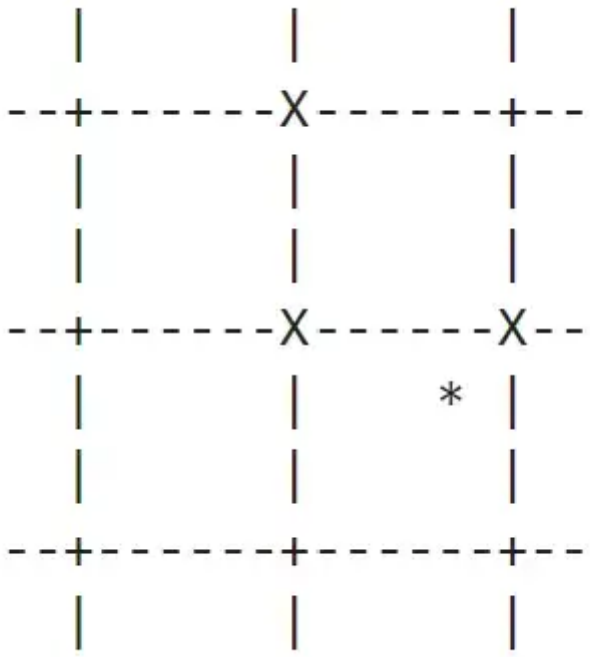
To make the problem cheaper computationally, the YOLOv7 loss finds first the anchor boxes that are likely to match each target box and treats them differently — these are known as the **center prior** anchor boxes. This process is applied at each FPN head, for each target box, across all images in batch at once.

Each anchor — which are the coordinates in our grid — defines a grid cell; where we consider the anchor to be at the top left of its corresponding grid cell. Subsequently, each cell (except cells on the border) has 4 adjacent cells (top, bottom, left, right). Each target box, for each FPN head, lies somewhere inside a grid cell. Imagine that we have the following grid, and the centre of a target box is represented by a `*`:



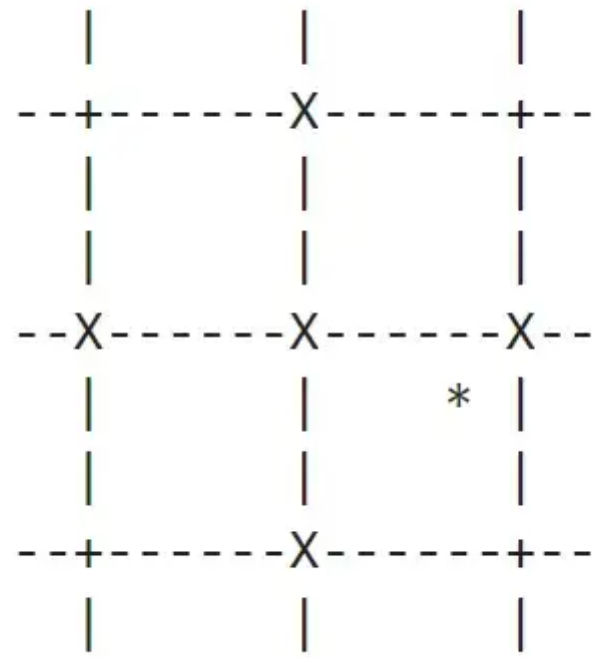
Based on the way the model is designed and trained, the `x` and `y` corrections that it can output are in the range of `[-0.5, 1.5]` grid cells. Thus, only a subset of the closest anchor boxes will be able to match the target centre. We select some of these anchor boxes to represent the **center prior** for the target box.

- For the Lead Heads, we use a **fine Center Prior**, which is a more targeted selection. This is comprised of **3 anchors per head:** the anchor associated the cell containing the target box centre, alongside the anchors for the 2 closest grid cells to the target box centre. In the diagram, the Center Prior anchors are marked with an `X`.



Selected centre priors for lead detection heads

- For the Auxiliary Heads (for models that use deep supervision), we use a **coarse Center Prior**, which is a less targeted selection. This is comprised of **5 anchors per head:** the anchor of the cell containing the target box centre, alongside all 4 adjacent grid cells.



Selected centre priors for auxillary detection heads

The reasoning behind this fine and coarse distinction is that the learning ability of Auxiliary Heads is lower than that of the Lead Heads, because the Lead Heads are deeper in the network. Thus, we try to avoid limiting too much from where the Auxiliary Head can learn to make sure we do not lose valuable information.

Similarly to the coordinate corrections, the model can only apply a multiplicative modifier to the width and height of each anchor box in the interval `[0, 4]`. This means that, at most, it can make the sides of the anchor boxes 4 times bigger. Therefore, from the anchor boxes selected as Center Prior, we filter those that are either 4 times bigger or smaller than the target box.

In summary, the Center Prior is comprised by the anchor boxes whose anchor is close enough to the target box centre and whose sides are not too far off from the target box side size.

Optimal Transport Assignment

One of the difficulties when evaluating object detection models is being able to match predicted boxes to target boxes in order to quantify if the model is doing a good job or not.

The simplest approach is to define an Intersection over Union (IoU) threshold and decide based on that. While this generally works, it becomes problematic when there are occlusions, ambiguity or when multiple objects are very close together. Optimal Transport Assignment (OTA) aims to solve some of these problems by considering label assignment as a global optimization problem for each image.

The main intuition consists in considering each target box a supplier of `k` positive label assignments and each predicted box a demander of either one positive label assignment or one background assignment. `k` is dynamic and depends on each target box. Then, transporting one positive label assignment from target box to predicted box has a cost based on classification and regression. Finally, the goal is to find a transportation plan (label assignment) that minimizes the total cost over the image.

This can be done using an off-the-shelf solver, but YOLOv7 implements **simOTA** (introduced in the YOLOX paper), a simplified version of the OTA problem. With the goal of reducing the computational cost of label assignment, it assigns the k predicted boxes for each target that have the lowest transportation cost instead of

solving the global problem. The Center Prior boxes are used as candidates for this process.

This helps us to further filter the amount of model outputs that can potentially be matched to a ground truth target.

YOLOv7 Loss algorithm

Now that we have introduced the most complicated pieces used in the YOLOv7 loss calculation, we can break down the algorithm used into the following steps:

1. For each FPN head (or each FPN head and Aux FPN head pair if Aux heads used):

- Find the Center Prior anchor boxes.
- Refine the candidate selection through the simOTA algorithm. Always use lead FPN heads for this.
- Obtain the **objectness loss** score using Binary Cross Entropy Loss between the predicted objectness probability and the Complete Intersection over Union (CIoU) with the matched target as ground truth. If there are no matches, this is 0.
- If there are any selected anchor box candidates, also calculate (otherwise they are just 0):
  - The **box (or regression) loss**, defined as the `mean(1 - CIoU)` between all candidate anchor boxes and their matched target.
  - The **classification loss**, using Binary Cross Entropy Loss between the predicted class probabilities for each anchor box and a one-hot encoded vector of the true class of the matched target.
- If model uses auxiliary heads, add each component obtained from the aux head to the corresponding main loss component (i.e., `x = x + aux_wt*aux_x`). The contribution weight (`aux_wt`) is defined by a predefined hyperparameter.
- Multiply the objectness loss by the corresponding FPN head weight (predefined hyperparameter).

2. Multiply each loss component (objectness, classification, regression) by their contribution weight (predefined hyperparameter).

3. Sum the already weighted loss components.

4. Multiply the final loss value by the batch size.

As a technical detail, the loss reported during evaluation is made computationally cheaper by skipping the simOTA and never using the auxiliary heads, even for the models that fashion deep supervision.

Whilst this process contains a lot of complexity, in practice, this is all encapsulated in a single class, which can be created as demonstrated below:

```
from yolov7 import create_yolov7_loss
```

```
loss_fn = create_yolov7_loss(model,
    image_size=640,
    box_loss_weight=0.05,
    cls_loss_weight=0.3,
    obj_loss_weight=0.7,
    ota_loss=True,)
```

Finetuning a model

Now that we understand how to use a pretrained model to make predictions, and how our loss function measures the quality of these predictions, let's look at how we can finetune a model to a custom task. To obtain the level of performance reported in the paper, YOLOv7 was trained using a variety of techniques. However, for our purposes, lets start with the minimal possible training loop required, before gradually introducing different techniques.

To handle the boilerplate aspects of the training loop, let's use PyTorch-accelerated. This will enable us to define only the the parts of the training loop which are relevant to our use case, without having to manage all of the boilerplate. To do this, we can override parts of the default PyTorch-accelerated `Trainer` and create a trainer specific to our YOLOv7 model, as demonstrated below:

```
                    ),
                    image_preds[:, 4:],
                    image_id.repeat(image_preds.shape[0])[None].T,
                ),
                1,
            )
        )

    if not formatted_preds:
        # if no predictions, create placeholder so that it can be gathered
        stacked_preds = torch.tensor(
            [self.YOLO7_PADDING_VALUE] * 7, device=self.device
        )[None]
    else:
        stacked_preds = torch.vstack(formatted_preds)

    return stacked_preds
```

Our training step is quite straightforward, with the only modification being that we need to extract the total loss from the dictionary that is returned. For the evaluation step, we first calculate the losses, and then retrieve the detections.

Evaluation logic

To evaluate our model's performance on this task, we can use Mean Average Precision (mAP); a standard metric for object detection tasks. Perhaps the most widely used (and trusted) implementation of mAP, is the class that is included in the PyCOCOTools package, which is used to evaluate official COCO leaderboard submissions.

However, as this does not have the most inituitive interface, we have created a simple wrapper around this, to make it a little more user-friendly. Additionally, as for many cases outside the COCO competition leaderboard, it can be advantageous to evaluate predictions using a fixed IoU threshold — as opposed to the range of IoUs that is used by default — we have added an option to do this to our evaluator.

To encapsulate our evaluation logic to use during training, let's create a callback for this; which will be updated at the end of each evaluation step and then calculated at the end of each evaluation epoch.

```
            distributed evaluation on a dataset where the batch size does not even
            """
            image_ids = labels[:, -1].tolist()

            # remove any image_idx that has already been seen
            # this can arise from distributed training where batch size does not ev
            seen_id_mask = torch.as_tensor(
                [False if idx not in self.image_ids else True for idx in image_ids]
            )

            if seen_id_mask.all():
                # no update required as all ids already seen this pass
                return []
            elif seen_id_mask.any():  # at least one True
                # remove predictions for images already seen this pass
                labels = labels[~seen_id_mask]

            return labels

        def _update(self, predictions):
            filtered_predictions = self._remove_seen(predictions)

            if len(filtered_predictions) > 0:
                self.eval_predictions.extend(filtered_predictions.tolist())
                updated_ids = filtered_predictions[:, -1].unique().tolist()
                self.image_ids.update(updated_ids)

        def _reset(self):
            self.image_ids = set()
            self.eval_predictions = []

        def _save_predictions(self, trainer, predictions_json):
            if (
                self.save_predictions_path is not None
                and trainer.run_config.is_world_process_zero
            ):
                with open(self.save_predictions_path / "predictions.json", "w") as
                    json.dump(predictions_json, f)
```

Now, all that we have to do is plug our callback into our Trainer, and our mAP will be recorded at each epoch!

Run training

Now, let's put everything we have seen so far into a simple training script. Here, we have used a simple training recipe that works well for a variety of tasks and have carried out minimal hyperparameter tuning.

As we noticed that the ground truth boxes for this dataset can contain quite a bit of space around the object, we decided to set the IoU threshold used for evaluation quite low; as it is likely that the boxes produced by the model will be tighter around the object.

```
if __name__ == "__main__":
    main()
```

Launching training as described here, using a single V100 GPU with fp16 enabled, after 3 epochs we obtained a mAP of `0.995`, which suggests that the model has learned the task almost perfectly!

However, whilst this is a great result, it is largely expected as COCO contains image of cars.

Training from scratch

Now that we have successfully finetuned a pretrained YOLOv7 model, let's explore how we can train the model from scratch. Whilst this could be done using numerous different training recipes, let's take a look at some of the key techniques that were used by the authors when training on COCO.

Mosaic Augmentation

Data augmentation is an important technique in deep learning where we synthetically expand our dataset by applying a series of augmentations to our data during training. Whilst common transforms in object detection tend to be augmentations such as flips and rotations, the YOLO authors take a slightly different approach by applying Mosaic augmentation; which was previously used by YOLOv4, YOLOv5 and YOLOX models.

The objective of mosaic augmentation is to overcome the observation that object detection models tend to focus on detecting items towards the centre of the image. The key idea is that, if we stitch multiple images together, the objects are likely to be in positions and contexts that are not normally observed in images seen in the dataset; which should force the features learned by the model to be more position invariant.

Whilst there are a couple of different implementations of mosaic, each with minor differences, here we shall present an implementation that combines four different images. This implementation has worked well for us in the past, with a variety of object detection models.

Although there is no requirement to resize images prior to creating a mosaic, it does result in the created mosaics being similar sizes. Therefore, we shall take that approach here. We can do this by creating a simple resizing transform and adding it to our dataset adaptor.

```
# https://github.com/Chris-hughes10/Yolov7-training/blob/main/yolov7/dataset.py

import albumentations as A

def create_base_transforms(target_image_size):
    return A.Compose(
        [
            A.LongestMaxSize(target_image_size),
        ],
        bbox_params=A.BboxParams(format="pascal_voc", label_fields=["labels"]),
    )
```

```
from yolov7.dataset import create_base_transforms
```

```
ds = CarsDatasetAdaptor(images_path, train_df,
                        transforms=create_base_transforms(640)
                       )
```

To apply our augmentations, once again, we are using Albumentations, which supports many object detection transforms.

Whilst data augmentations are usually implemented as functions, which are passed to a PyTorch dataset and applied shortly after loading an image, as mosaic requires loading multiple images from the dataset, this approach will not work here. We decided to implement mosaic as a dataset wrapper class, to cleanly encapsulate this logic. We can import and use this as demonstrated below:

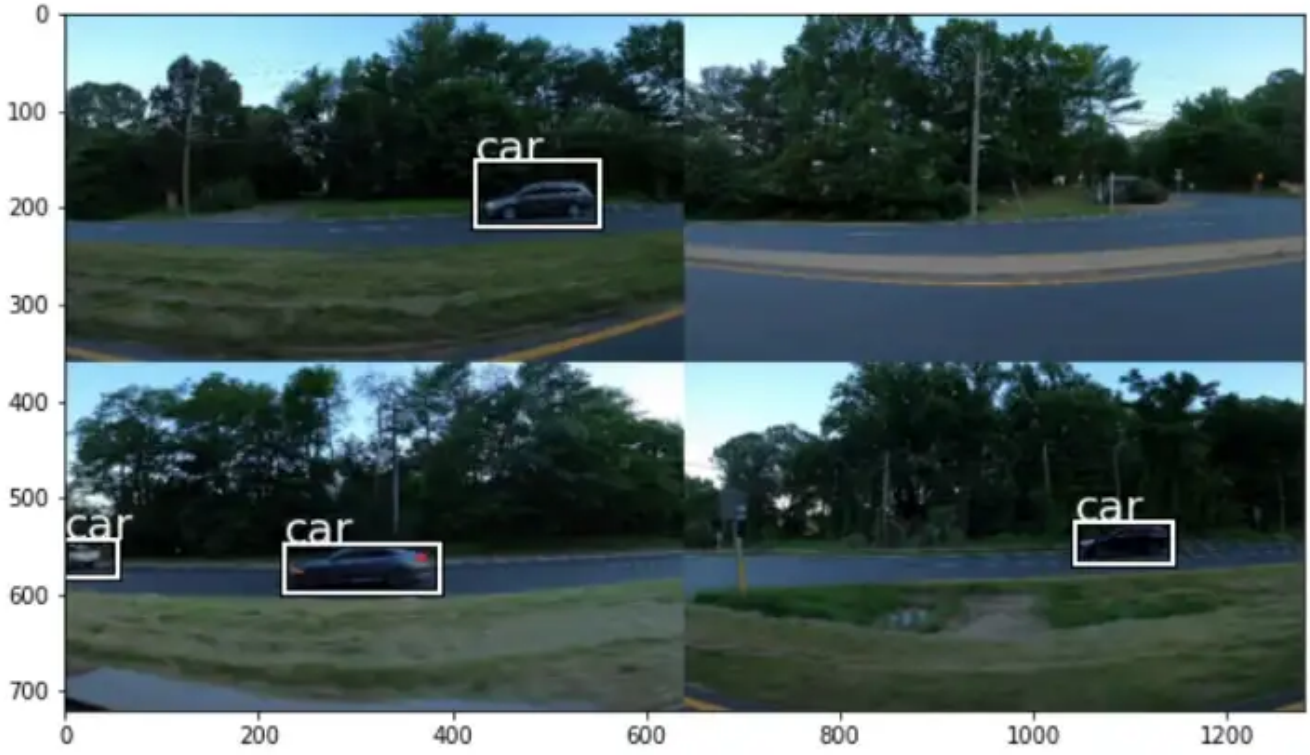
```
from yolov7.mosaic import MosaicMixupDataset
```

```
mds = MosaicMixupDataset(ds, apply_mosaic_probability=1, apply_mixup_probability=0)
```

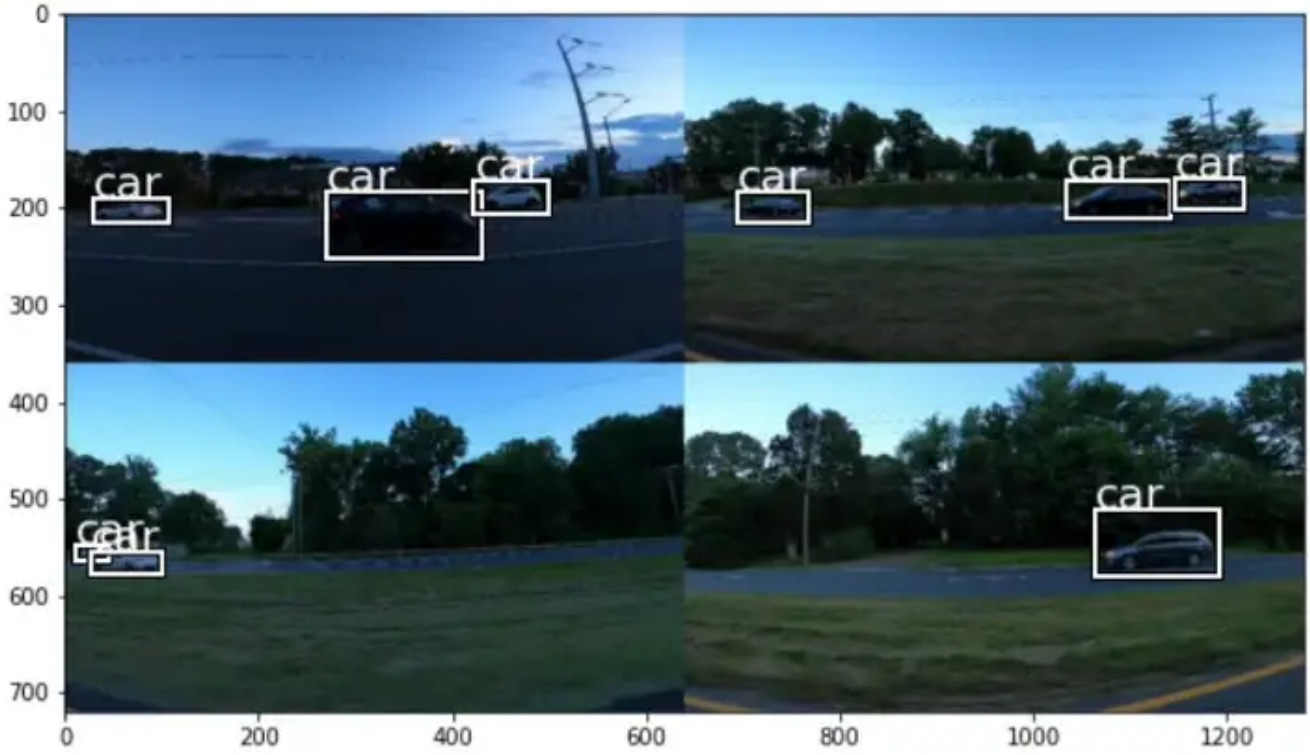
Let's take a look at some examples of the types of images that are produced. As we haven't (yet) passed any resizing transforms to our mosaic dataset, these images are quite large.

```
def plot_mosaic(idx):
    image, xyxy_bboxes, class_ids, image_ids, image_size = mds[idx]
    show_image(image, xyxy_bboxes.tolist(), [lookups['class_id_to_label'][int(c)] for c in class_ids])
    print(f'Image id: {image_idx}')
    print(f'Original Image size: {image_size}')
    print(f'Image size: {image.shape}')
```

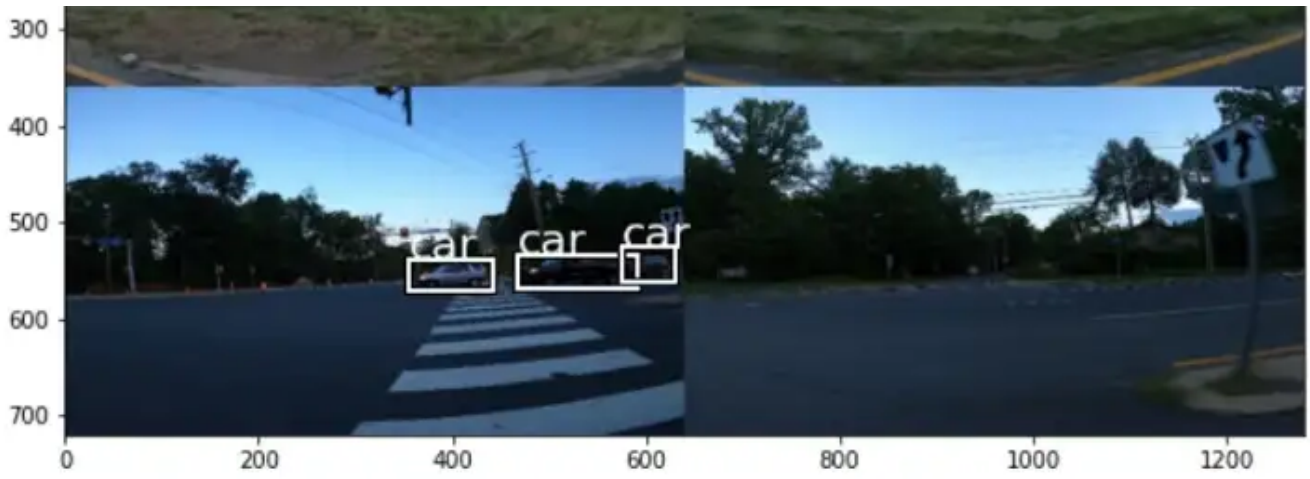
```
for i in range(3):
    plot_mosaic(4)
```




```
Image id: 6
Original Image size: (720, 1280)
Image size: (720, 1280, 3)
```




```
Image id: 6
Original Image size: (720, 1280)
Image size: (720, 1280, 3)
```




```
Image id: 6
Original Image size: (720, 1280)
Image size: (720, 1280, 3)
```

training — the model just sees a continuous stream of images either way!

Mixup Augmentation

Mosaic augmentation is often applied alongside another transform — Mixup. To visualise what this does, let's disable mosaic for the moment and enable mixup on its own, we can do this as demonstrated below:

```
mds.enable(apply_mosaic_probability=0, apply_mixup_probability=1)
```

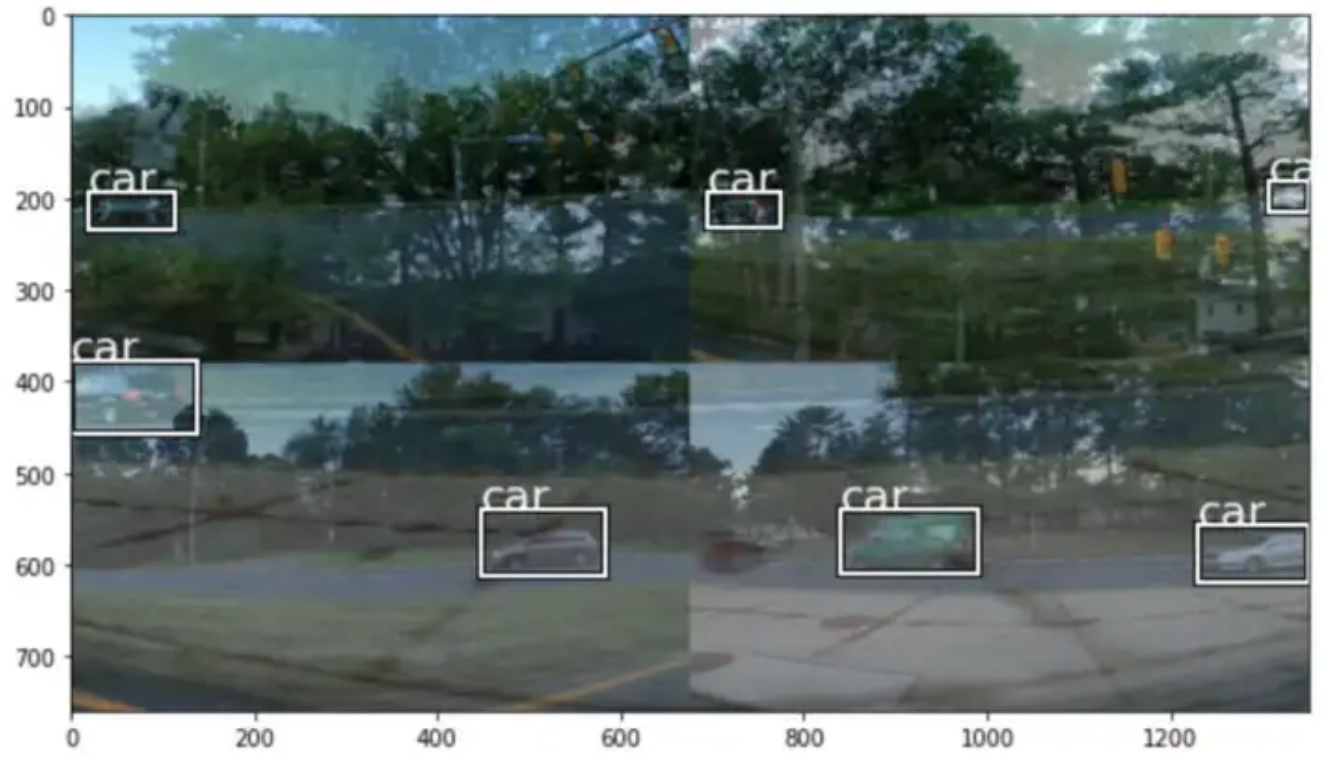
```
plot_mosaic(4)
```


```
Image id: 6
Original Image size: (380, 676)
Image size: (360, 640, 3)
```

Interesting! We can see that it has combined two images together, which results in some 'ghostly' looking cars and backgrounds! Now, let's enable both transforms and inspect our outputs.

```
mds.enable(apply_mosaic_probability=1, apply_mixup_probability=1)
```

```
plot_mosaic(4)
```




```
Image id: 6
Original Image size: (760, 1352)
Image size: (760, 1352, 3)
```

Wow! There are quite a lot of cars to detect in our resulting image, in many different positions — which will definitely be a challenge for the model! Notice that when we apply mosaic and mixup together, a single image is mixed with a mosaic.

Post-mosaic affine transforms

As we noted earlier, the mosaics that we are creating are significantly bigger than the image sizes we will use to train our model, so we will need to do some sort of resizing here. The simplest way would be to simply apply a resize transform after creating the mosaic.

Whilst this would work, this is likely to result in some very small objects, as we are essentially resizing four images to the size of one - which is likely to become a problem where the domain already contains very small bounding boxes!

Additionally, each of our mosaics are structurally quite similar, with an image in each quadrant. Recalling that our aim was to make the model more robust to position changes, this may not actually help that much; as the model is likely just to start looking in the middle of each quadrant.

To overcome this, one approach that we can take is to simply take a random crop from our mosaic. This will still provide the variability in positioning whilst preserving the size and aspect ratio of the target objects. At this point, it may also be a good opportunity to add in some other transforms such as scaling and rotation to add even more variability.

The exact transforms, and magnitudes, used will be heavily dependent on the images that you are using, so we would recommend experimenting with these setting first — to ensure that all objects are still visible and recognisable — prior to training a model!

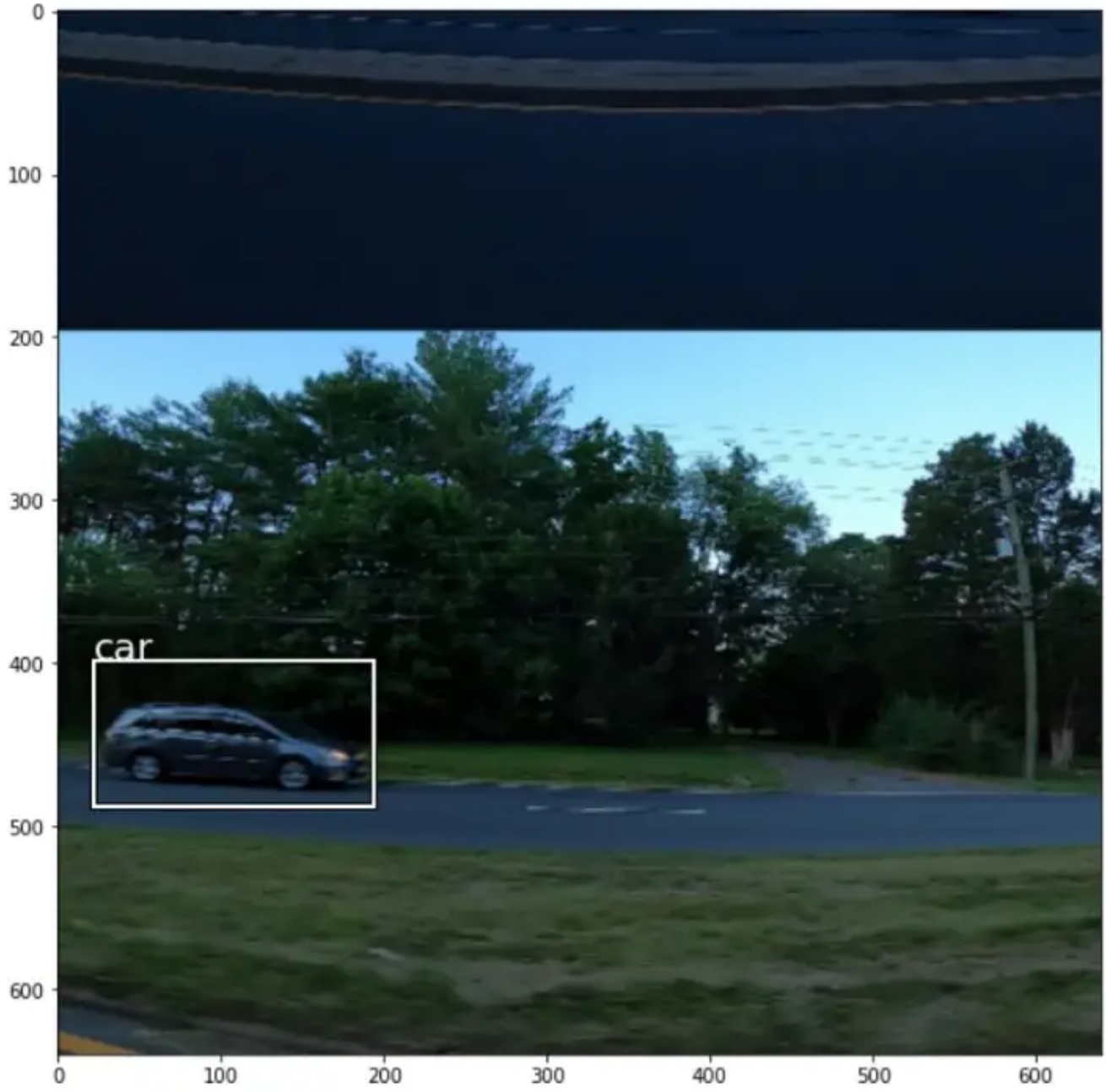
We can define the transforms to apply to our mosaic images as demonstrated below. Here, we have chosen a selection of affine transforms — in sensible ranges for our target data — followed by a random crop. Following the original implementation, we are also applying mixup less frequently than mosaic.

```
            ),
            A.HorizontalFlip(),
            A.RandomResizedCrop(height=output_height, width=output_width, scale
        ],
        bbox_params=A.BboxParams(format="pascal_voc", label_fields=["labels"],
    )
```

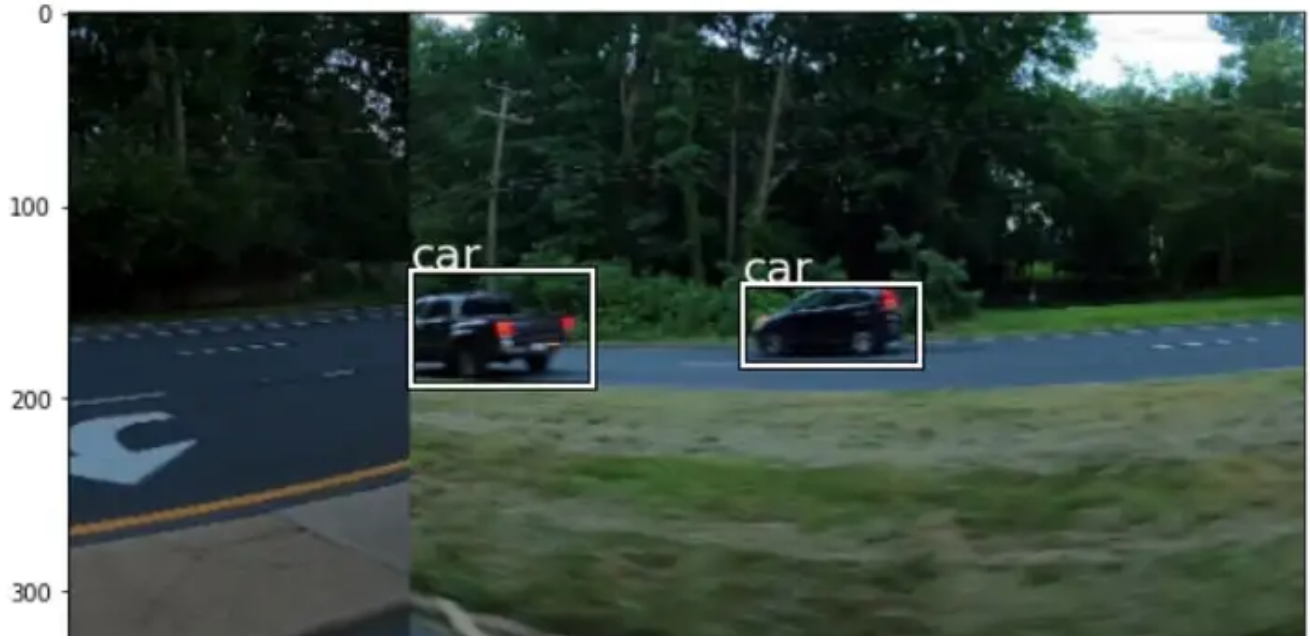
```
from yolov7.mosaic import create_post_mosaic_transform
```

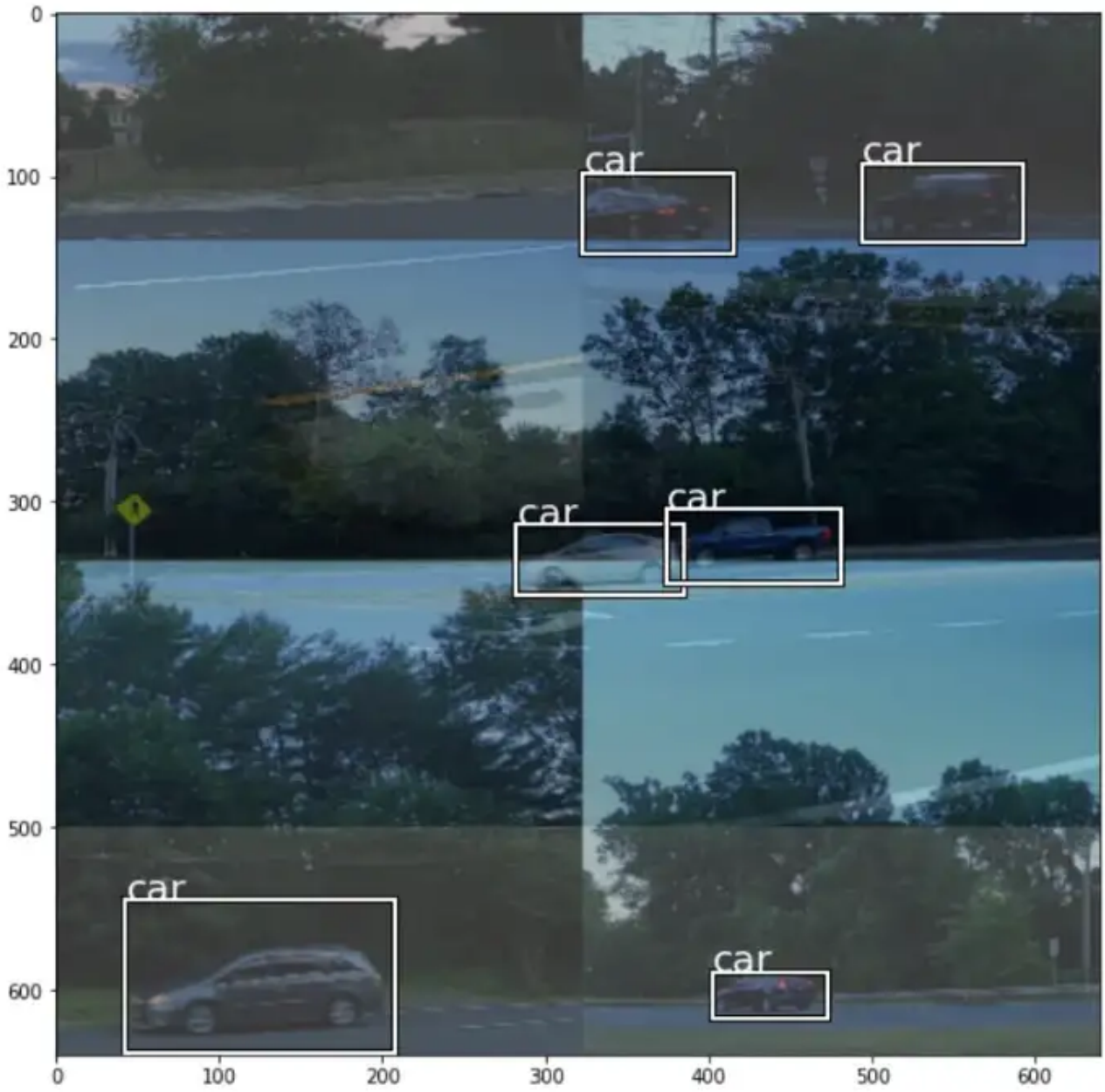
```
mds = MosaicMixupDataset(ds, apply_mixup_probability=0.2, post_mosaic_transforms=create_post_mosaic_transform(640, 640))
```

```
for i in range(5):
    plot_mosaic(4)
```

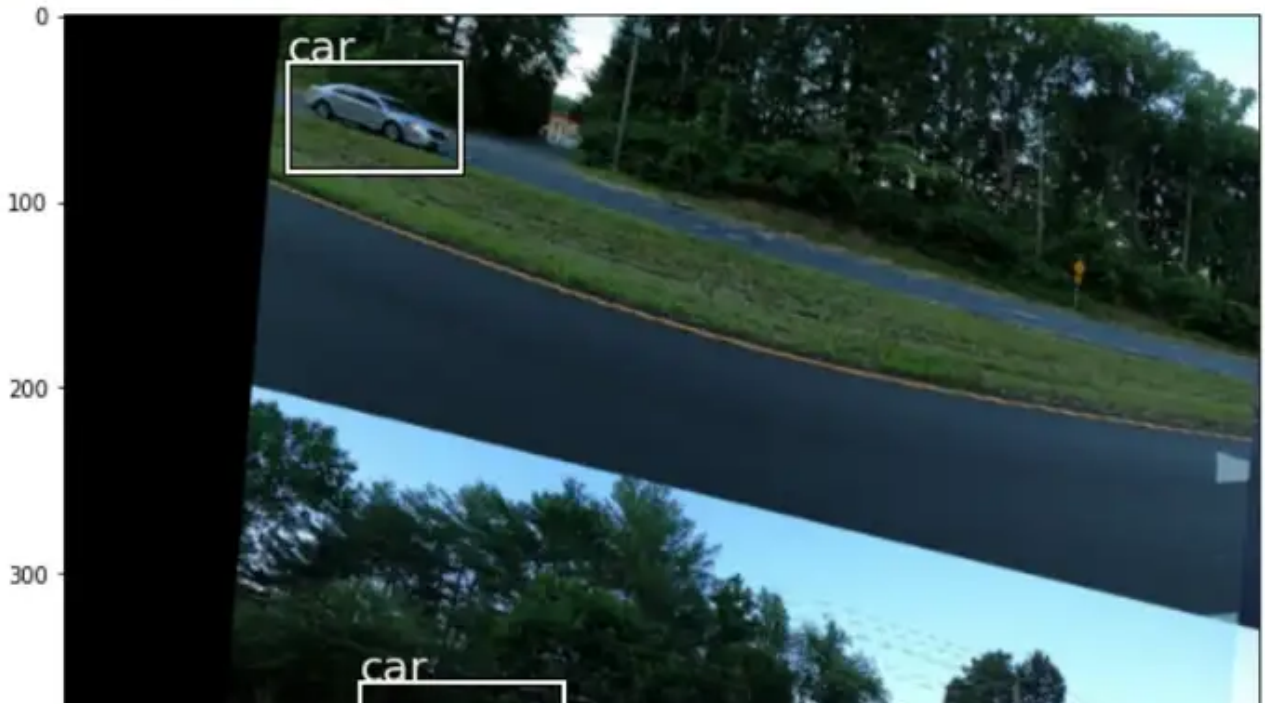


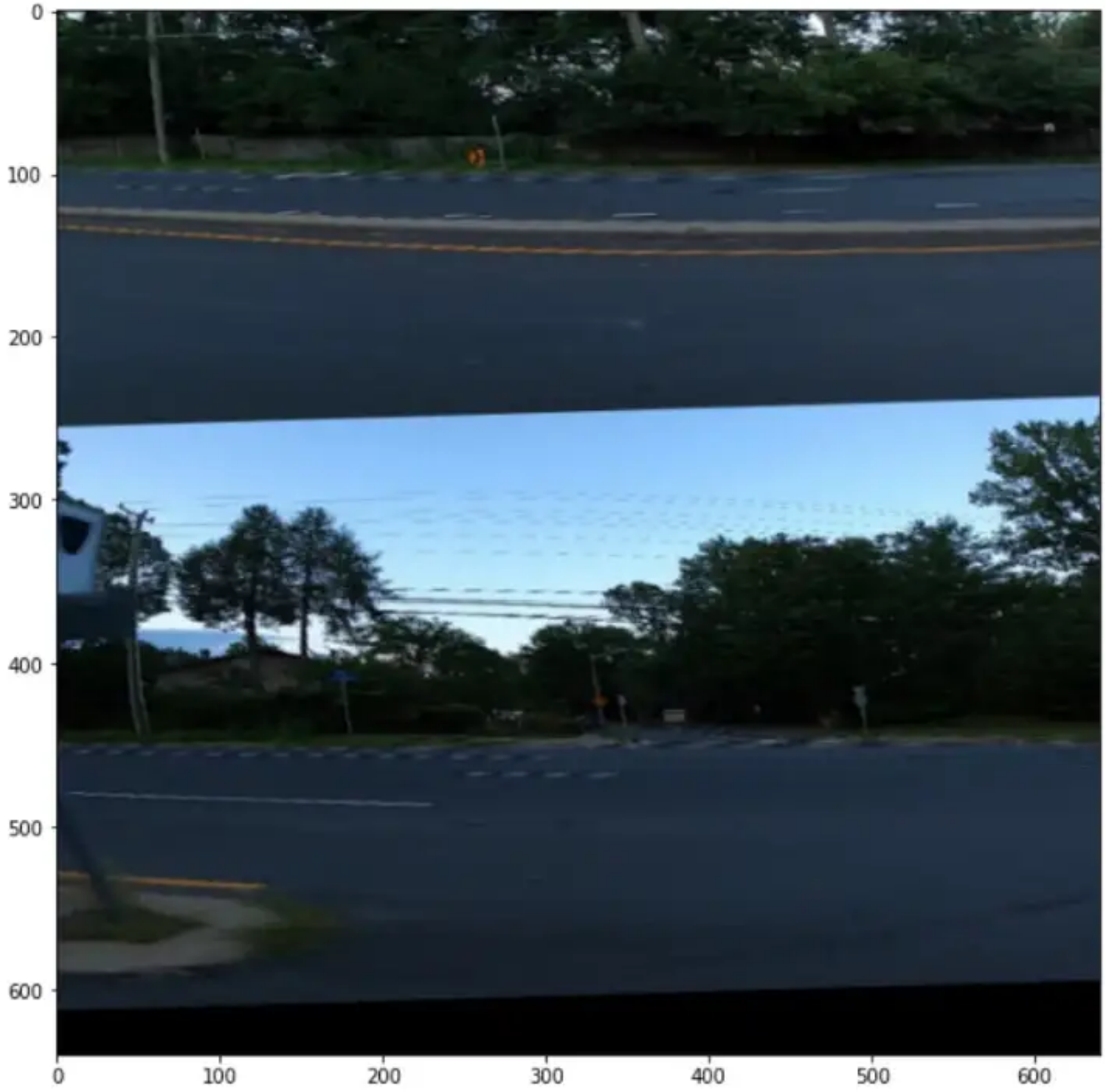

```
Image id: 6
Original Image size: (760, 1352)
Image size: (640, 640, 3)
```






```
Image id: 6
Original Image size: (760, 1352)
Image size: (640, 640, 3)
```






```
Image id: 6
Original Image size: (760, 1352)
Image size: (640, 640, 3)
```

Looking at these images, we can see a huge amount of variation and the images are now the correct size for training. As we have selected a random scale, we can also see that not every image looks like a mosaic, so these outputs should not be too dissimilar to the images that the model will see during inference. If more extreme augmentations are used — such that there is a notable difference between the training and inference images — it can be advantageous to disable these shortly before the end of training.

In the official implementation, the authors use mosaics of both 4 and 9 images during training. However, inspecting the outputs of these augmentations when

combined with scaling and cropping, in many cases the outputs looked very similar, so we have chosen to omit this here.

Applying weight decay to parameter groups

In our simple example earlier, we created our optimizer so that it would optimize all of the parameters of our model. However, if we would like to follow the authors in introducing weight decay regularization, following the guidance given in Bag of Tricks for Image Classification with Convolutional Neural Networks this may not be optimal; with this paper recommending that weight decay should be applied to only convolutional and fully connected layers.

To implement this in PyTorch, we will need to create two distinct parameter groups to be optimized; one containing our convolutional weights and the other with the remaining parameters. We can do this as demonstrated below:

```
param_groups = model.get_parameter_groups()
param_groups.keys()
```

```
dict_keys(['conv_weights', 'other_params'])
```

Inspecting the method definition, we can see that this is a simple filter operation:

```
??model.get_parameter_groups
```

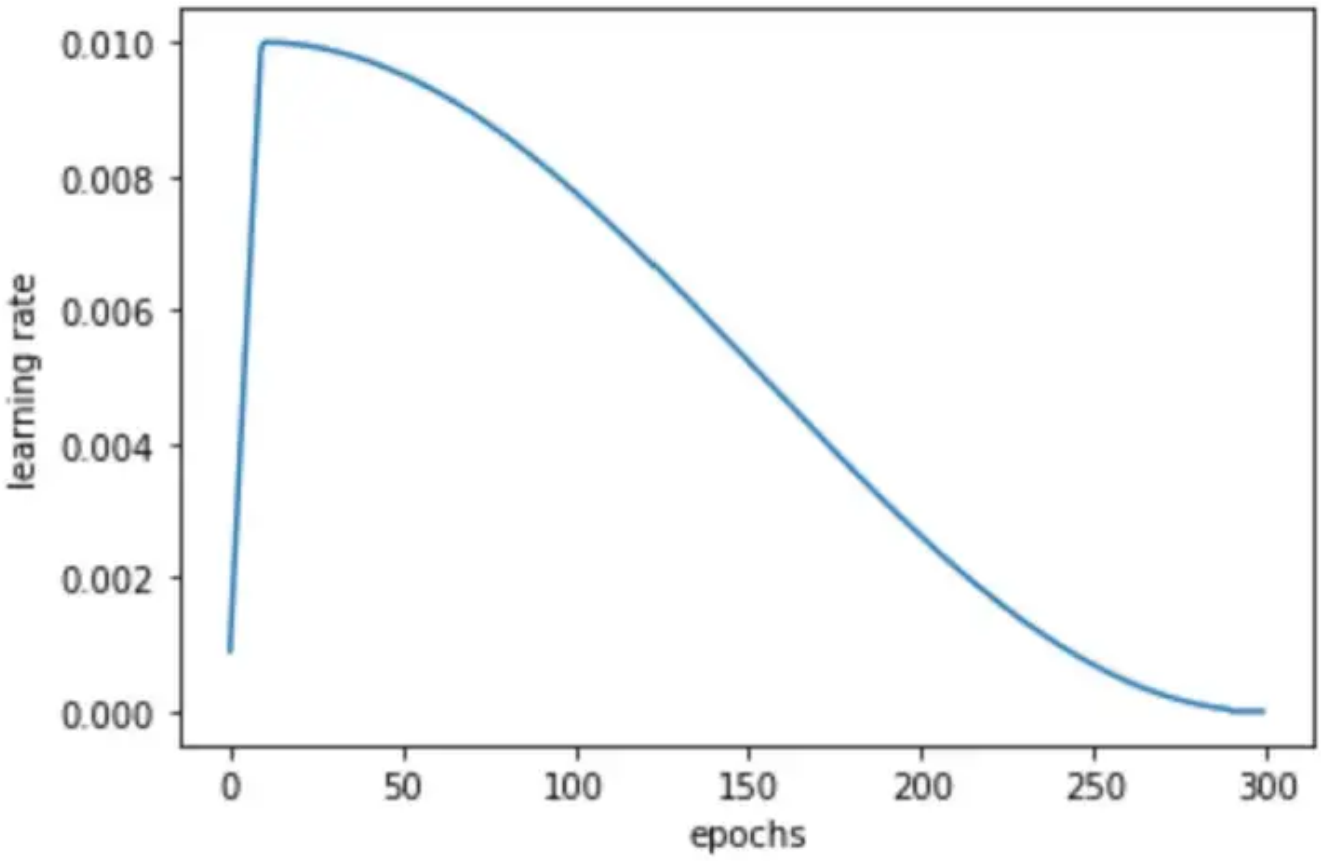
```
Signature: model.get_parameter_groups()
Docstring: <no docstring>
Source:
    def get_parameter_groups(self):
        conv_weights = {
            v.weight
            for k, v in self.model.named_modules()
            if (
                hasattr(v, "weight")
                and isinstance(v.weight, nn.Parameter)
                and not isinstance(v, nn.BatchNorm2d)
            )
        }

        other_params = [p for p in self.model.parameters() if p not in conv_weights]

        return {"conv_weights": list(conv_weights), "other_params": other_params}
File:      /mnt/yolov7/models/yolo.py
Type:      method
```

Now we can simply pass these to the optimizer:

```
optimizer = torch.optim.SGD(
        param_groups["other_params"], lr=0.01, momentum=0.937, nesterov=True
    )

optimizer.add_param_group(
        {"params": param_groups["conv_weights"], "weight_decay": weight_decay}
    )
```

Learning rate scheduling

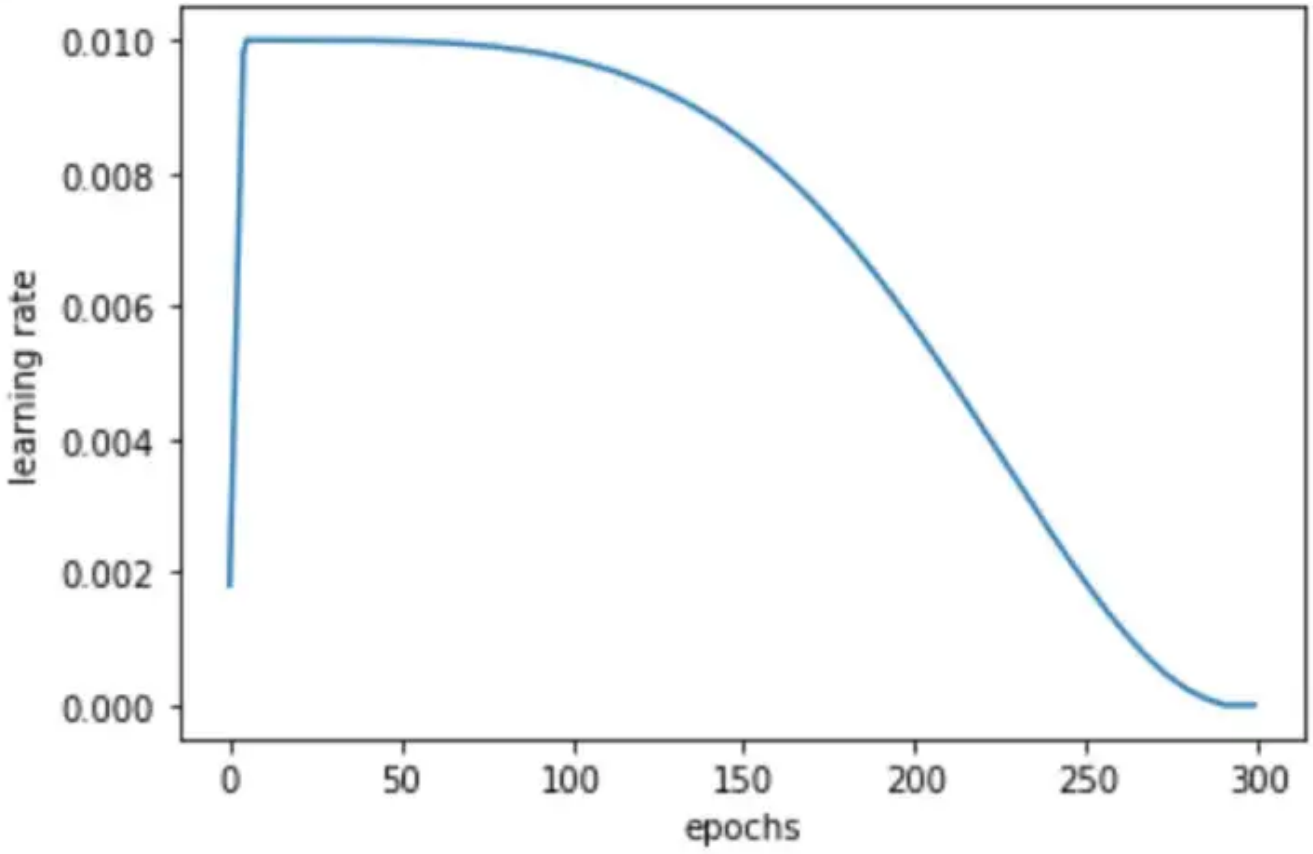
When training neural networks, we often wish to adjust the value of our learning rate during training; this is done using a learning rate scheduler. Whilst there are many popular schedules, the authors opt for a cosine learning rate schedule — with a linear warmup at the start of training. This has the following shape:



Cosine learning rate schedule (with warmup)

In practice, we find that a period of warmup, and cooldown — where the learning rate is held at its minimum value — is often a good strategy for this scheduler. Additionally, the scheduler PyTorch-accelerated supports a k-decay argument which can be used to adjust how aggressive the annealing is.

For this problem, we found that using k-decay to hold the learning rate at a higher value for longer worked quite well. This schedule, along with warmup and cooldown epochs, can be seen below:



Cosine learning rate schedule (with warmup), set with k_decay = 2

Gradient accumulation, scaling weight decay

When training a model, the batch size we use is often determined by our hardware; as we want to try to maximise the amount of data that we can put on the GPU. However, some considerations must be made:

- For very small batch sizes, we are unable to approximate the gradients of the whole dataset. This can result in unstable training.
- Modifying the batch size can result in different settings being needed for hyperparameters such as the learning rate and weight decay. This can make it difficult to find a consistent set of hyperparameters.

To overcome this, the authors use use a technique called *gradient accumulation*, in which the gradients from multiple steps are accumulated to simulate a bigger batch size. For example, suppose that the maximum batch size that we can fit on our GPU is 8. Instead of updating the parameters of the model at the end of each batch, we can save gradient values, proceed to the next batch and add these new gradients. After a designated number of steps, we then perform the update; if we set our number of steps to 4, this is roughly equivalent of using a batch size of 32!

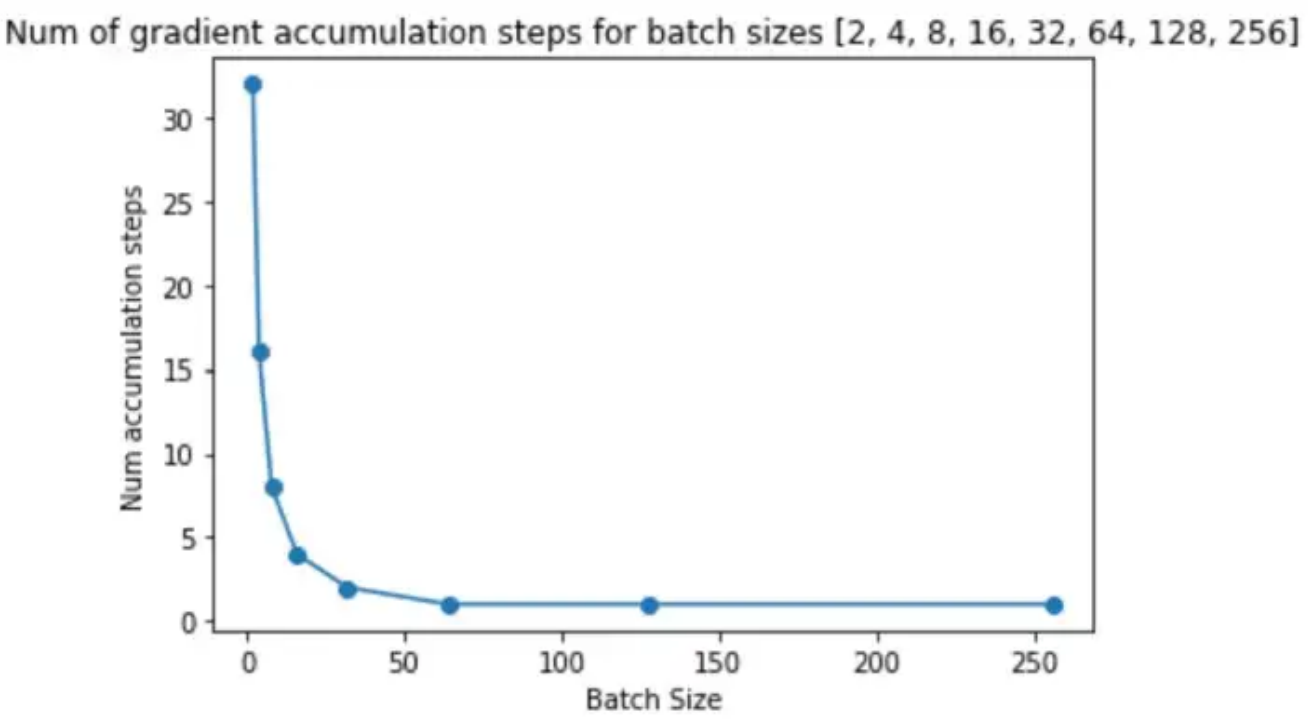
In PyTorch, this could be performed manually as follows:

```
num_accumulation_steps = 4

# loop through ennumerated batches
for step, (inputs, labels) in enumerate(data_loader):

        model_outputs = model(inputs)
        loss  = loss_fn(model_outputs, labels)

        # normalize loss to account for batch accumulation
        loss = loss / num_accumulation_steps

        # calculate gradients, these are summed automatically
        loss.backward()

        if ((step + 1) % num_accumulation_steps == 0) or
            (step + 1 == len(data_loader)):
            # perform weight update
            optimizer.step()
            optimizer.zero_grad()
```

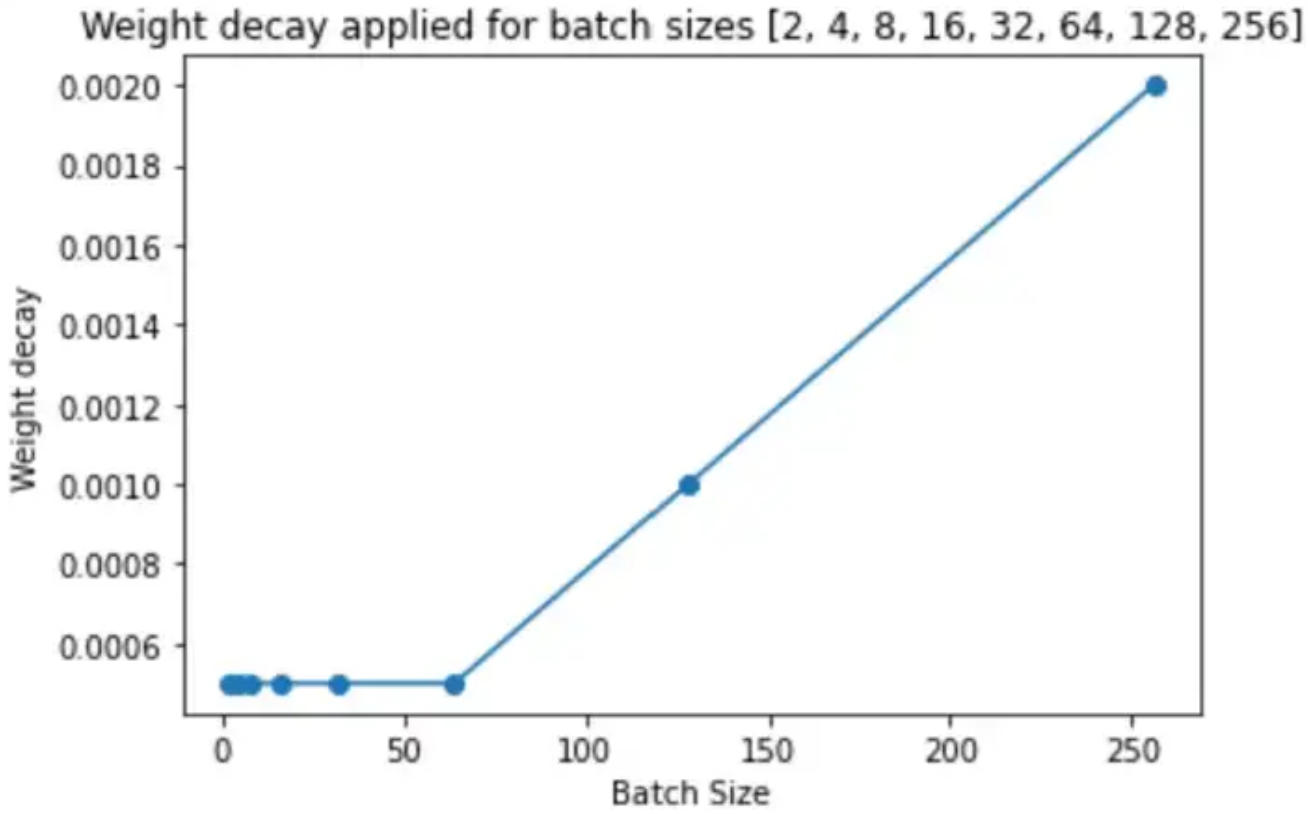
In the original YOLOv7 implementation, the number of gradient accumulation steps is selected so that the total batch size (across all processes) is at least 64; which mitigates both of the issues discussed earlier. Additionally, the authors scale the weight decay used based on the batch size in the following way:

```
nominal_batch_size = 64
num_accumulate_steps = max(round(nominal_batch_size / total_batch_size), 1)

base_weight_decay = 0.0005
scaled_weight_decay = (
    base_weight_decay * total_batch_size * num_accumulate_steps / nominal_batch
)
```

We can visualise these relationships below:



Looking first at the number of accumulation steps, we can see that the number of accumulation steps decreases until we hit our nominal batch size, and then gradient accumulation is no longer needed.



Now looking at the amount of weight decay used, we can see that it is held at the base value until the nominal batch size is reached, and then is scaled linearly with

the batch size; with more weight decay applied as the batch size gets bigger.

Model EMA

When training a model, it can be beneficial to set the values for the model weights by taking a moving average of the parameters that were observed across the entire training run, as opposed to using the parameters obtained after the last incremental update. This is often done by maintaining an exponentially weighted average (EMA) of the model parameters, in practice, this usually means maintaining another copy of the model to store these averaged weights. However, rather than updating all of the parameters of this model after every update step, we set these parameters using a linear combination of the existing parameter values and the updated values.

This is done using the following formula:

```
updated_EMA_model_weights = decay * EMA_model_weights + (1. - decay) * updated_
```

where the *decay* is a parameter that we set. For example, if we set decay=0.99, we have:

```
updated_EMA_model_weights = 0.99 * EMA_model_weights + 0.01 * updated_model_we
```

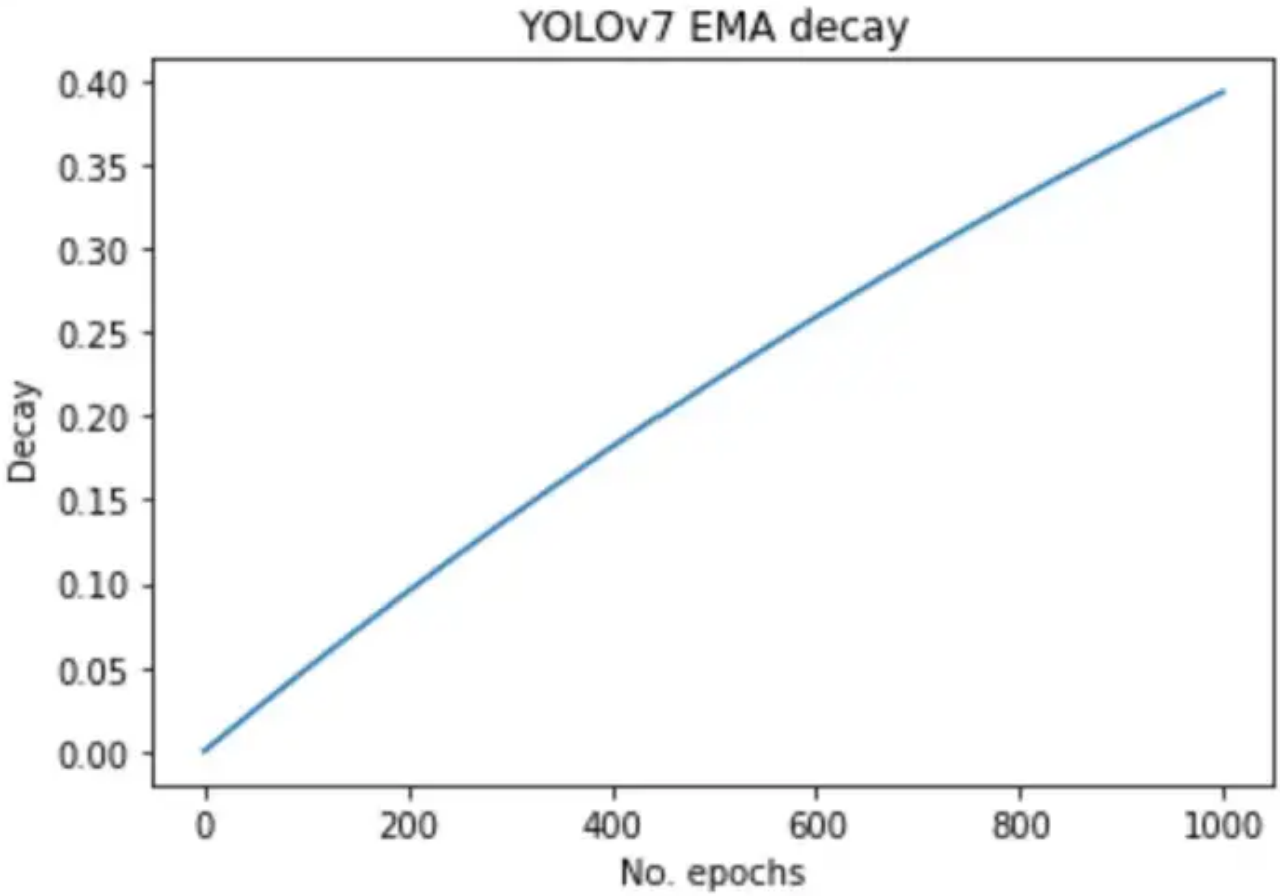
which we can see is keeping 99% of the existing state and only 1% of the new state!

To understand why this may be beneficial, let's consider the case that our model, in an early stage of training, performs exceptionally poorly on a batch of data. This may result in a large update update to our parameters, overcompensating for the high loss obtained, which will be detrimental for the upcoming batches. By only incorporating only a small percentage of the latest parameters, large updates will be 'smoothed', and have less of an overall impact on the model's weights. Sometimes, these averaged parameters can sometimes produce significantly better results during evaluation, and this technique has been employed in several training schemes for popular models such as training MNASNet, MobileNet-V3 and EfficientNet; using the implementation included in TensorFlow.

The approach to EMA taken by the YOLOv7 authors is slightly different to other implementations as, instead of using a fixed decay, the amount of decay changes based on the number of updates that have been made. We can extend the ModelEMA class included with PyTorch-accelerated to implement this behaviour as defined below:

```
# https://github.com/Chris-hughes10/Yolov7-training/blob/main/yolov7/utils.py

from pytorch-accelerated.utils import ModelEma

class Yolov7ModelEma(ModelEma):
    def __init__(self, model, decay=0.9999):
        super().__init__(model, decay)
        self.num_updates = 0
        self.decay_fn = lambda x: decay * (
            1 - math.exp(-x / 2000)
        )  # decay exponential ramp (to help early epochs)
        self.decay = self.decay_fn(self.num_updates)

    def update(self, model):
        super().update(model)
        self.num_updates += 1
        self.decay = self.decay_fn(self.num_updates)
```

Here, we can see that the decay is set by calling a function after each update. Let's visualise how this looks:



From this, we can see that the amount of decay increases with the number of updates, which is once per epoch.

Recalling the formulas above, this means that, initially, we favour using the updated model weights rather than a historical average. However, as training progresses, we start to incorporate more of the averaged weights from previous epochs. This is an interesting departure from the usual usage of this technique, which is designed to help the EMA model converge more quickly in earlier epochs.

Selecting appropriate anchor box sizes

Recalling the earlier discussion on anchor boxes, and how these play an important part on how YOLOv7 is able to detect objects, let's look at how we can evaluate whether our chosen anchor boxes are suitable for our problem and, if not, find some sensible choices for our dataset.

The approach here is largely adapted from the autoanchor approach used in YOLOv5, which was also used with YOLOv7.

Evaluating current anchor boxes

The simplest approach would be to simply use the same anchor boxes as used for COCO, which are already bundled with the defined architectures.

```
from yolov7 import create_yolov7_model
```

```
model = create_yolov7_model('yolov7', pretrained=False)
```

```
model.detection_head.anchor_grid
```

```
tensor([[[[[[ 12.,  16.]]],


          [[[ 19.,  36.]]],


          [[[ 40.,  28.]]]]],



        [[[[[ 36.,  75.]]],


          [[[ 76.,  55.]]],


          [[[ 72., 146.]]]]],



        [[[[[142., 110.]]],


          [[[192., 243.]]],


          [[[459., 401.]]]]])
```

Here we can see that we have 3 groups, one for each layer of the feature pyramid network. The numbers correspond to our anchor sizes, the width and height of the anchor boxes that will be generated.

Recall that, the Feature Pyramid Network (FPN) has three outputs, and each output's role is to detect objects according to their scale.

For example:

- P3/8 is for detecting smaller objects.
- P4/16 is for detecting medium objects.
- P5/32 is for detecting bigger objects.

With this in mind, we need to set our anchor sizes accordingly for each layer.

```
current_anchors = model.detection_head.anchor_grid.clone().cpu().view(-1, 2); current_anchors
```

```
tensor([[ 12.,  16.],
        [ 19.,  36.],
        [ 40.,  28.],
        [ 36.,  75.],
        [ 76.,  55.],
        [ 72., 146.],
        [142., 110.],
        [192., 243.],
        [459., 401.]])
```

To evaluate our current anchor boxes, we can calculate the best possible recall, which would occur if the model was able to successfully match an appropriate anchor box with a ground truth.

Find and Resize ground truth bounding boxes

To evaluate our anchor boxes, we first need some knowledge of the shapes and sizes of the objects in our dataset. However, before we can evaluate, we need to resize the width and height of our ground truth boxes based on the size of the images that we will train on — for this architecture, this is recommended to be 640.

Let's start by finding the width and height of all ground truth boxes in the training set. We can calculate these as demonstrated below:

```
train_annotations_df = train_df.query('has_annotation == True').copy()
```

```
train_annotations_df['h'] = train_annotations_df['ymax'] -  train_annotations_df['ymin']
train_annotations_df['w'] = train_annotations_df['xmax'] -  train_annotations_df['xmin']
```

```
train_annotations_df
```

	image	xmin	ymin	xmax	ymax	class_name	has_annotation	image_id	class_id	h	w
0	vid_4_1000.jpg	281.259045	187.035071	327.727931	223.225547	car	True	0	0.0	36.190476	46.468886
1	vid_4_10000.jpg	15.163531	187.035071	120.329957	236.430180	car	True	1	0.0	49.395109	105.166425
2	vid_4_10040.jpg	239.192475	176.764801	361.968162	236.430180	car	True	3	0.0	59.665380	122.775687
4	vid_4_10060.jpg	16.630970	186.546010	132.558611	238.386422	car	True	4	0.0	51.840412	115.927641
5	vid_4_10100.jpg	447.568741	160.625804	582.083936	232.517696	car	True	6	0.0	71.891892	134.515195
...	...	...	...	...	...	...	...	...	...	...	...
554	vid_4_9860.jpg	0.000000	198.321729	49.235251	236.223284	car	True	994	0.0	37.901554	49.235251
555	vid_4_9880.jpg	329.876184	156.482351	536.664239	250.497895	car	True	995	0.0	94.015544	206.788055
556	vid_4_9900.jpg	0.000000	168.295823	141.797524	239.176652	car	True	996	0.0	70.880829	141.797524
557	vid_4_9960.jpg	487.428988	172.233646	616.917699	228.839864	car	True	999	0.0	56.606218	129.488711
558	vid_4_9980.jpg	221.558631	182.570434	348.585579	238.192196	car	True	1000	0.0	55.621762	127.026948

397 rows × 11 columns

```
raw_gt_wh = train_annotations_df[['w', 'h']].values
```

Next, we will need the height and width of our images. Sometimes, we have this information ahead of time, in which case we can use this knowledge directly. Otherwise, we can do this as follows:

```
from PIL import Image
from tqdm.contrib.concurrent import process_map
```

```
def find_image_size(image_path):
    image = Image.open(image_path)
    w, h = image.size
    return (image_path.parts[-1], (w, h))

image_sizes = process_map(find_image_size, [images_path/p for p in train_df.image.unique()])
```

```
100%|██████████| 324/324 [00:00<00:00, 5619.21it/s]
```

```
image_sizes_df = pd.DataFrame(dict(image_sizes)).T.reset_index().rename(columns={'index': 'image', 0: 'image_w', 1:'image_h'})
```

```
image_sizes_df
```

	image	image_w	image_h
0	vid_4_1000.jpg	676	380
1	vid_4_10000.jpg	676	380
2	vid_4_10040.jpg	676	380
3	vid_4_10060.jpg	676	380
4	vid_4_10100.jpg	676	380
...	...	...	...
319	vid_4_13060.jpg	676	380
320	vid_4_13100.jpg	676	380
321	vid_4_13240.jpg	676	380
322	vid_4_13280.jpg	676	380
323	vid_4_13300.jpg	676	380

324 rows × 3 columns

We can now merge this with our existing DataFrame:

```
train_annotations_df = pd.merge(train_annotations_df, image_sizes_df, on='image'); train_annotations_df
```

	image	xmin	ymin	xmax	ymax	class_name	has_annotation	image_id	class_id	h	w	image_w	image_h
0	vid_4_1000.jpg	281.259045	187.035071	327.727931	223.225547	car	True	0	0.0	36.190476	46.468886	676	380
1	vid_4_10000.jpg	15.163531	187.035071	120.329957	236.430180	car	True	1	0.0	49.395109	105.166425	676	380
2	vid_4_10040.jpg	239.192475	176.764801	361.968162	236.430180	car	True	3	0.0	59.665380	122.775687	676	380
3	vid_4_10060.jpg	16.630970	186.546010	132.558611	238.386422	car	True	4	0.0	51.840412	115.927641	676	380
4	vid_4_10100.jpg	447.568741	160.625804	582.083936	232.517696	car	True	6	0.0	71.891892	134.515195	676	380
...	...	...	...	...	...	...	...	...	...	...	...	...	...
392	vid_4_9860.jpg	0.000000	198.321729	49.235251	236.223284	car	True	994	0.0	37.901554	49.235251	676	380
393	vid_4_9880.jpg	329.876184	156.482351	536.664239	250.497895	car	True	995	0.0	94.015544	206.788055	676	380
394	vid_4_9900.jpg	0.000000	168.295823	141.797524	239.176652	car	True	996	0.0	70.880829	141.797524	676	380
395	vid_4_9960.jpg	487.428988	172.233646	616.917699	228.839864	car	True	999	0.0	56.606218	129.488711	676	380
396	vid_4_9980.jpg	221.558631	182.570434	348.585579	238.192196	car	True	1000	0.0	55.621762	127.026948	676	380

397 rows × 13 columns

```
image_sizes = train_annotations_df[['image_w', 'image_h']].values
```

Now, we can use this information to get the resized widths and heights of our ground truth targets, with respect to our target image size. To preserve the aspect ratios of the objects in our images, the recommended approach to resizing is to

scale the image so that the longest size is equal to our target size. We can do this using the function below:

```
# https://github.com/Chris-hughes10/Yolov7-training/blob/main/yolov7/anchors.py

def calculate_resized_gt_wh(gt_wh, image_sizes, target_image_size=640):
    """
    Given an array of bounding box  widths and heights, and their correspondin
    resize these relative to the specified target image size.

    This function assumes that resizing will be performed by scaling the image
    side is equal to the given target image size.

    :param gt_wh: an array of shape [N, 2] containing the raw width and height
    :param image_sizes: an array of shape [N, 2] or [1, 2] containing the widt
    :param target_image_size: the size of the images that will be used during t

    """
    normalized_gt_wh = gt_wh / image_sizes
    target_image_sizes = (
        target_image_size * image_sizes / image_sizes.max(1, keepdims=True)
    )

    resized_gt_wh = target_image_sizes * normalized_gt_wh

    tiny_boxes_exist = (resized_gt_wh < 3).any(1).sum()
    if tiny_boxes_exist:
        print(
            f"""WARNING: Extremely small objects found.
            {tiny_boxes_exist} of {len(resized_gt_wh)} labels are < 3 pixels in
            """
        )
        resized_gt_wh = resized_gt_wh[(resized_gt_wh >= 2.0).any(1)]

    return resized_gt_wh
```

```
from yolov7.anchors import calculate_resized_gt_wh
```

```
raw_gt_wh.shape
```

```
(397, 2)
```

```
gt_wh = calculate_resized_gt_wh(raw_gt_wh, image_sizes, target_image_size=640); gt_wh[:5]
```

```
array([[ 43.99421122,  34.26317273],
       [ 99.56584662,  46.76460062],
       [116.23733718,  56.48793344],
       [109.75397973,  49.07967981],
       [127.35166419,  68.06332961]])
```

Alternatively, as all of our images are the same size in this case, we could simply specify a single image size.

```
calculate_resized_gt_wh(raw_gt_wh, image_sizes=np.array([[676, 380]]), target_image_size=640)[:5]
```

```
array([[ 43.99421122,  34.26317273],
       [ 99.56584662,  46.76460062],
       [116.23733718,  56.48793344],
       [109.75397973,  49.07967981],
       [127.35166419,  68.06332961]])
```

Note that we have also filtered out any boxes what will be incredibly small (less than 3 pixels in either height or width), with respect to the new image size, as these boxes are usually too small to be considered useful!

Calculating Best Possible Recall

Now that we have the width and height of all ground truth boxes in our training set, we can evaluate our current anchor boxes as follows:

```
        best_possible_recall = (
            (best_anchor_ratio > 1.0 / LOSS_ANCHOR_MULTIPLE_THRESHOLD).float().mean
        )

        return best_possible_recall
```

```
calculate_best_possible_recall(current_anchors, gt_wh)
```

```
tensor(1.)
```

From this, we can see that the current anchor boxes are a good fit for this dataset; which makes sense, as the images are quite similar to those in COCO.

How does this work?

At this point, you may be wondering, how exactly do we calculate the best possible recall. To answer this, let's go through the process manually.

Intuitively, we would like to ensure that at least one anchor can be matched to each ground truth box. Whilst we could do this by framing it as an optimization problem — how do we match each ground truth box with its optimal anchor — this would introduce a lot of complexity for what we are trying to do.

Given an anchor box, we need a simpler way of measuring how well it can be made to fit a ground truth box. Let's examine one approach that can be taken to do this, starting with the width and height of a single ground truth box.

```
gt_box_wh = gt_wh[0]; gt_box_wh
```

```
array([43.99421122, 34.26317273])
```

For each anchor box, we can inspect the ratios of its height and width when compared to the height and width of our ground truth target and use this to understand where the biggest differences are.

```
current_anchors/gt_box_wh
```

```
tensor([[ 0.2728,  0.4670],
        [ 0.4319,  1.0507],
        [ 0.9092,  0.8172],
        [ 0.8183,  2.1889],
        [ 1.7275,  1.6052],
        [ 1.6366,  4.2611],
        [ 3.2277,  3.2104],
        [ 4.3642,  7.0922],
        [10.4332, 11.7035]], dtype=torch.float64)
```

As the scale of these ratios will depend on whether the anchor box sides are greater or smaller than the sides of our ground truth box, we can ensure that our magnitudes are in the range [0, 1] by also calculating the reciprocal and taking the minimum ratios for each anchor.

```
symmetric_size_ratios = torch.min(current_anchors/gt_box_wh, gt_box_wh/current_anchors); symmetric_size_ratios
```

```
tensor([[0.2728, 0.4670],
        [0.4319, 0.9518],
        [0.9092, 0.8172],
        [0.8183, 0.4568],
        [0.5789, 0.6230],
        [0.6110, 0.2347],
        [0.3098, 0.3115],
        [0.2291, 0.1410],
        [0.0958, 0.0854]], dtype=torch.float64)
```

From this, we now have an indication of how well, independently, the width and height of each anchor box ‘fits’ to our ground truth target.

Now, our challenge is how to evaluate the matching of the the width and height together!

One way we can approach this is, to take the minimum ratio for each anchor; representing the side that worst matches our ground truth.

```
worst_side_size_ratio = symmetric_size_ratios.min(-1).values; worst_side_size_ratio
```

```
tensor([0.2728, 0.4319, 0.8172, 0.4568, 0.5789, 0.2347, 0.3098, 0.1410, 0.0854],
       dtype=torch.float64)
```

The reason why we have selected the worst fitting side here, is because we know that the other side matches our target *at least* as well as the one selected; we can think of this as the worst case scenario!

Now, let's select the anchor box which matches the best out of these options, this is simply the largest value.

```
best_anchor_ratio = worst_side_size_ratio.max(-1).values; best_anchor_ratio
```

```
tensor(0.8172, dtype=torch.float64)
```

Out of the worst fitting options, this is our selected match!

Recalling that the loss function only looks to match anchor boxes that are up to 4 times greater or smaller than the size of the ground truth target, we can now verify whether this anchor is within this range and would be considered a successful match.

We can do that as demonstrated below, taking the reciprocal of our loss multiple, to ensure that it is in the same range as our value:

```
from yolov7.anchors import LOSS_ANCHOR_MULTIPLE_THRESHOLD

LOSS_ANCHOR_MULTIPLE_THRESHOLD
```

```
4
```

```
best_anchor_ratio > 1. / LOSS_ANCHOR_MULTIPLE_THRESHOLD
```

```
tensor(True)
```

From this, we can see that at least one of our anchors could be successfully matched to our selected ground truth target!

Now that we understand the sequence of steps, we can now apply the same logic to all of our ground truth boxes to see how many matches we can obtain with our current set of anchors:

```
symmetric_size_ratios = torch.min(current_anchors[None]/gt_wh[:, None], gt_wh[:, None]/current_anchors[None])

worst_side_size_ratio = symmetric_size_ratios.min(-1).values
best_anchor_ratio = worst_side_size_ratio.max(-1).values
```

```
best_anchor_ratio > 1. / LOSS_ANCHOR_MULTIPLE_THRESHOLD
```

```
tensor([True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True, True, True, True, True, True, True, True, True, True, True, True,
        True])
```

Now that we have calculated, for each ground truth box, whether it has a match. We can take the mean number of matches to find out best possible recall; in our case, this is 1, as we saw earlier!

```
best_possible_recall = (best_anchor_ratio > 1. / LOSS_ANCHOR_MULTIPLE_THRESHOLD).float().mean(); best_possible_recall
```

```
tensor(1.)
```

Selecting new anchor boxes

Whilst using the pre-defined anchors may be a good choice for similar datasets, this may not be appropriate for all datasets, for example, those that contain lots of small objects. In these cases, a better approach may be to select entirely new anchors.

Let's explore how we can do this!

First, let's define the number of anchors that we need for our architecture.

```
num_anchors = current_anchors.shape[0]; num_anchors
```

```
9
```

Now, based on our bounding boxes, we need to define a sensible set widths and heights of anchor templates. One way that we can estimate this is by using K-means to cluster our ground truth aspect ratios, based on the number of anchor sizes that we need. We can then use these centroids as our starting estimates. We can do this using the following function:

```
# https://github.com/Chris-hughes10/Yolov7-training/blob/main/yolov7/anchors.py

def estimate_anchors(num_anchors, gt_wh):
    """
    Given a target number of anchors and an array of widths and heights for eac
    estimate a set of anchors using the centroids from Kmeans clustering.

    :param num_anchors: the number of anchors to return
    :param gt_wh: an array of shape [N, 2] representing the width and height of

    """
    print(f"Running kmeans for {num_anchors} anchors on {len(gt_wh)} points..."
    std_dev = gt_wh.std(0)
    proposed_anchors, _ = kmeans(
        gt_wh / std_dev, num_anchors, iter=30
    )  # divide by std so they are in approx same range
    proposed_anchors *= std_dev

    return proposed_anchors
```

```
from yolov7.anchors import estimate_anchors
```

```
proposed_anchors = estimate_anchors(num_anchors, gt_wh); proposed_anchors
```

```
Running kmeans for 9 anchors on 397 points...
array([[157.29889337,  57.47936534],
       [ 70.37782144,  28.9259909 ],
       [117.67344588,  55.91906451],
       [ 71.90866699,  50.8658278 ],
       [186.28917826,  84.09961313],
       [ 38.3042379 ,  25.61764762],
       [ 56.74676871,  36.96437289],
       [112.83692506,  42.92073289],
       [ 90.20439918,  36.43405811]])
```

Here, we can see that we now have a set of anchor templates that we can use as a starting point. As before, let's calculate our best possible recall using these anchor boxes:

```
calculate_best_possible_recall(proposed_anchors, gt_wh)
```

```
tensor(1.)
```

Once again, we see that our best possible recall is 1, which means that these anchor sizes are also a good fit for our problem!

Whilst it is perhaps unnecessary in this case, we may be able improve these anchors further using a genetic algorithm. Following this methodology, we can define a *fitness* (or reward) function to measure how well our anchor boxes match our data and make small, random changes to our anchor sizes to try and maximise this function.

In this case we can define our fitness function as follows:

```
        best_anchor_ratio = calculate_best_anchor_ratio(anchors=anchors, gt_wh=wh)
        return (
            best_anchor_ratio
            * (best_anchor_ratio > 1 / LOSS_ANCHOR_MULTIPLE_THRESHOLD).float()
        ).mean()
```

Here, we are taking the best anchor ratio for each match that will be considered during the loss calculation. If an anchor box is more than four times greater or smaller than its matched bounding box, it will not contribute to our score. Let's use this to calculate a fitness score for our proposed anchor sizes:

```
from yolov7.anchors import anchor_fitness
```

```
anchor_fitness(proposed_anchors, gt_wh)
```

```
tensor(0.8825, dtype=torch.float64)
```

Now, let's use this as the fitness function when optimizing our anchors, as demonstrated below:

```
from yolov7.anchors import evolve_anchors
```

```
evolved_anchors = evolve_anchors(proposed_anchors, gt_wh, anchor_fitness_fn=anchor_fitness, num_iterations=30000); evolved_anchors
```

```
Evolving anchors with Genetic Algorithm: fitness = 0.8855: 100%|| 30000/30000 [00:19<00:00,
array([[156.06907735,  61.09621462],
       [ 66.80622862,  29.0438958 ],
       [136.33063134,  52.56100946],
       [ 80.06700492,  36.42821897],
       [179.52356295,  83.0822995 ],
       [ 37.64336168,  29.35155407],
       [ 52.00907081,  37.50677223],
       [114.96001811,  44.23679448],
       [ 99.66602472,  37.86939469]])
```

Inspecting the definition of this function, we can see that, for a specified number of iterations, we are simply sampling random noise from a normal distribution and using this to mutate our anchor sizes. If this change leads to an increased score, we keep these as our anchor sizes!

```
            )
            if verbose:
                print(f"Iteration: {i}, Fitness: {best_fitness}")

    return proposed_anchors
```

Let's see whether this has improved our score at all:

```
anchor_fitness(evolved_anchors, gt_wh)
```

```
tensor(0.8876, dtype=torch.float64)
```

We can see that our evolved anchors have a better fitness score than our original proposed anchors, as we would expect!

Now, all that is left to do is to sort the anchors into a rough ascending order, considering the smallest dimension for each anchor.

```
evolved_anchors = torch.as_tensor(evolved_anchors)[torch.sort(torch.as_tensor(evolved_anchors.min(-1))).indices]
```

```
calculate_best_possible_recall(evolved_anchors, gt_wh)
```

```
tensor(1.)
```

Putting it all together

Now that we understand the process, we could calculate our anchors for our dataset in a single step using the following function.

```
from yolov7.anchors import calculate_anchors
```

```
new_anchors = calculate_anchors(current_anchors, image_sizes, gt_wh, target_image_size=640, best_possible_recall_threshold=0.98); new_anchors
```

```
Best Possible Recall (BPR) = 1.0000
tensor([[ 12.,  16.],
        [ 19.,  36.],
        [ 40.,  28.],
        [ 36.,  75.],
        [ 76.,  55.],
        [ 72., 146.],
        [142., 110.],
        [192., 243.],
        [459., 401.]])
```

In this case, as our best possible recall is already greater than the threshold, we can keep our original anchor sizes!

However, in cases where our anchor sizes change, we can update them as demonstrated below:

```
model.update_anchors(new_anchors)
```

Run training

Now we have explored some of the tecnhiques used in the original training recipe, let's update our training script to include some of these features. An updated script is presented below:

```
if __name__ == "__main__":
    main()
```

Launching training once again, as described here, using a single V100 GPU with fp16 enabled, after 300 epochs we obtained a mAP of `0.997`, for both the model and the EMA model; a marginal increase over our transfer learning run, and probably the maximum performance that can be achieved on this dataset!

Conclusion

Hopefully that has provided a somewhat comprehensive overview of some of the most interesting ideas from the YOLOv7 training process, and how these can be applied in custom training scripts.

All of the code required to replicate this post is available as a notebook here. Whilst code snippets are used throughout the article, this is primarily for aesthetic purposes, please defer to the notebook, and the repo for working code.

Chris Hughes and Bernat Puig Camps are on LinkedIn

Dataset Used

Here, we used the car object detection dataset from Kaggle which was made publicly available as part of Competition Six (tjmachinelearning.com). This dataset is frequently used for learning purposes.

Whilst there is no clear license attached to this dataset, we received explicit permission from the authors to use this as part of this article. Unless otherwise stated, all images used in this article are taken from this dataset.

References

- [2207.02696] YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors (arxiv.org)
- Responding to the Controversy about YOLOv5 (roboflow.com)
- [2011.08036] Scaled-YOLOv4: Scaling Cross Stage Partial Network (arxiv.org)

- WongKinYiu/yolor: implementation of paper — You Only Learn One Representation: Unified Network for Multiple Tasks (https://arxiv.org/abs/2105.04206) (github.com)
- ultralytics/yolov5: YOLOv5 🚀 in PyTorch > ONNX > CoreML > TFLite (github.com)
- Chris-hughes10/Yolov7-training: A clean, modular implementation of the Yolov7 model family, which uses the official pretrained weights, with utilities for training the model on custom (non-COCO) tasks. (github.com)
- WongKinYiu/yolov7: Implementation of paper — YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors (github.com)
- Albumentations: fast and flexible image augmentations
- Non Maximum Suppression Explained | Papers With Code
- [1612.03144] Feature Pyramid Networks for Object Detection (arxiv.org)
- Jaccard index — Wikipedia
- [2103.14259] OTA: Optimal Transport Assignment for Object Detection (arxiv.org)
- [2107.08430] YOLOX: Exceeding YOLO Series in 2021 (arxiv.org)
- Chris-hughes10/pytorch-accelerated: A lightweight library designed to accelerate the process of training PyTorch models by providing a minimal, but extensible training loop which is flexible enough to handle the majority of use cases, and capable of utilizing different hardware options with no code changes required. Docs: https://pytorch-accelerated.readthedocs.io/en/latest/ (github.com)
- Trainer — pytorch-accelerated 0.1.3 documentation
- Evaluation measures (information retrieval) — Wikipedia
- pycocotools · PyPI
- Callbacks — pytorch-accelerated 0.1.3 documentation
- Quickstart — pytorch-accelerated 0.1.3 documentation

- [1710.09412v2] mixup: Beyond Empirical Risk Minimization (arxiv.org)
- Deep learning basics — weight decay | by Sophia Yang | Analytics Vidhya | Medium
- [1812.01187] Bag of Tricks for Image Classification with Convolutional Neural Networks (arxiv.org)
- What is Gradient Accumulation in Deep Learning? | by Raz Rotenberg | Towards Data Science
- Utils — pytorch-accelerated 0.1.3 documentation
- Genetic Algorithms — GeeksforGeeks

Deep Learning Yolov 7 Object Detection Deep Dives Editors Pick

Thanks to Ben Huberman