Alma Mater Studiorum - Università di Bologna
LM Informatica

YOLO v3 re-implementation for the project of the course "Machine Learning"

Marco Ferrati, Michele Perlino, Tommaso Azzalin

A.Y. 2021/2022

Abstract

This report is not complete, since the project was wider than the work on the YOLO v3 network. Only the chapter regarding YOLO v3 is kept.

Contents

Chapter 1

Network architecture: YOLO v3

1.1 YOLO Overview

YOLO is a Convolutional Neural Network (CNN) for performing object detection in real-time.
CNNs are classifier-based systems that can process input images as structured arrays of data and identify patterns between them.
To date, there are two main types of object detection algorithms in the field of deep learning:

- **Classification-based algorithms** (aka *Two Stage Detectors*): firstly, they select a group of *Region of Interest* (ROI) in the images where the chances that an object is present are high; secondly, they apply Convolutional Neural Networks (CNN) techniques to these selected regions to detect the presence of an object.
  - a problem associated with these types of algorithms is that they need to execute a detector in each ROI, and this makes the process of object detection very slow and highly expensive in terms of computation.

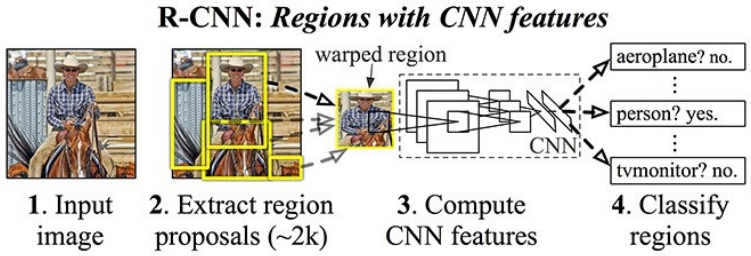

Figure 1.1: Detection using R-CNN (A classification based algorithm).

- **Regression-based algorithms** (aka *Single Stage Detectors*): these types of algorithms are faster then the above algorithms, in that there is no selection of the ROI, so that the bounding boxes and the labels are predicted for the whole image at once; they are able to identify and classify objects within the image at once.
  - beyond the higher speed, a key point is that the predictions are informed by the global context in the image, thus they generally lead to higher accuracies.

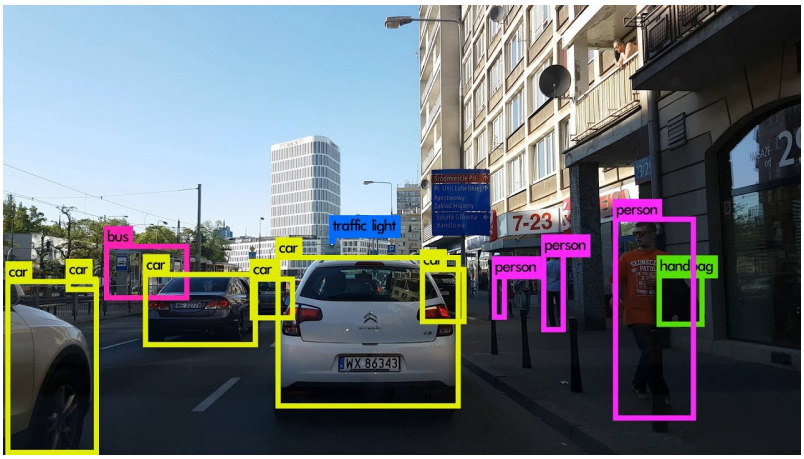

Figure 1.2: Detection using YOLO (A regression based algorithm).

YOLO, as its acronym reveals (You Only Look Once), falls into the regression-based algorithms: it applies a single neural network to the full image. In a nutshell, this network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities: in detail, each bounding box has a corresponding confidence score of how accurate it assumes that prediction should be and detects only one object per bounding box. The boundary boxes are generated by clustering the dimensions of the ground truth boxes from the original dataset to find the most common shapes and sizes.
Since it makes predictions with a single network evaluation, it is extremely fast, more than 1000x faster than *R-CNN* (which requires instead thousands of network evaluations for a single image) and 100x faster than *Fast R-CNN*.

1.1.1 YOLO history and milestones

YOLO is originally introduced by Joseph Redmon in Darknet: Darknet is an open source library written in C and CUDA (thus supports CPU and GPU computation) by Joseph Redmon himself in order to ease the process of implementing YOLO and other object detection models. YOLO's first version breaks the competition over R-CNN and DPM (Deformable Parts Model) thanks to:

- real-time frames processing at 45 FPS;
- less false positives on the background;
- higher detection accuracy (although lower accuracy on localization).

The algorithm has continued to evolve ever since its initial release in 2016. Both YOLO v2 and YOLO v3 were written by Joseph Redmon. After YOLO v3, there came new authors who anchored their own goals in every other YOLO release.
In summary:

1. **YOLO v2**: released in 2017, this version earned an honorable mention at CVPR 2017 because of significant improvements on anchor boxes, batch normalization and higher resolution.

2. **YOLO v3**: the 2018 release had an additional objectness score to the bounding box prediction and connections to the backbone network layers. It also provided an improved performance on tiny objects because of the ability to run predictions at three different levels of granularity.

3. **YOLO v4**: April's release of 2020 became the first paper not authored by Joseph Redmon. Here Alexey Bochkovski introduced novel improvements, including mish activation, improved feature aggregation, etc.

4. **YOLO v5**: Glenn Jocher continued to make further improvements in his June 2020 release, focusing on the architecture itself: however, this release is not published yet, it is still considered in progress.

We decided to re-implement in PyTorch the 3rd version since it is well-documented and widely tested by several computer vision communities: there are not many works and experiments with regard to the 4th version yet and there is not a scientific paper regarding the 5th version.

1.1.2 YOLO architectures at a glance

YOLO v2 used a custom deep architecture *darknet-19*[1], an originally 19-layer network supplemented with 11 more layers for object detection. With a 30-layer architecture, YOLO v2 often struggled with small object detections: this was attributed to loss of fine-grained features as the layers downsampled the input. To remedy this, YOLO v2 used an identity mapping, concatenating feature maps from a previous layer to capture low level features. However, YOLO v2's architecture was still lacking some of the most important elements that are now staple in most of state-of-the art algorithms, such as residual blocks, skip connections and upsampling: YOLO v3 incorporates all of these. First, YOLO v3 uses *Darknet-53* as its backbone[2], which, as the name reveals, has a 53 layer network trained on Imagenet: for the task of detection, 53 more layers are stacked onto it, leading to a 106 layer fully convolutional underlying architecture for YOLO v3. This is the reason behind the slowness of YOLO v3 compared to YOLO v2. The table below shows its higher efficiency with reference to competing backbones (ResNet-101 or ResNet-152).

Backbone	Top-1	Top-5	Ops	BFLOP/s	FPS
Darknet-19	74.1	91.8	7.29	1246	**171**
ResNet-101	77.1	93.7	19.7	1039	53
ResNet-152	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

Figure 1.3: Comparison of backbones. Accuracy, billions of operations (Ops), billion floating-point operations per second (BFLOP/s), and frames per second (FPS) for various networks.

Darknet-53 is 1.5 times faster than ResNet-101 and 2 times faster than ResNet-152, without entailing any trade-off in terms of accuracy and speed, since it is still as accurate as ResNet-152.
YOLO v3 is fast and accurate in terms of mean average precision (mAP) and intersection over union (IoU) values as well. It runs significantly faster than other detection methods with comparable performance.

[1] *Darknet-n* refers to a convolutional neural network which is n layers deep.

[2] Backbone is not a universal technical term in deep learning and it has mainly two meaning: it can refer to the feature extraction part of the network or, more trivially, to the crucial part of the overall architecture.

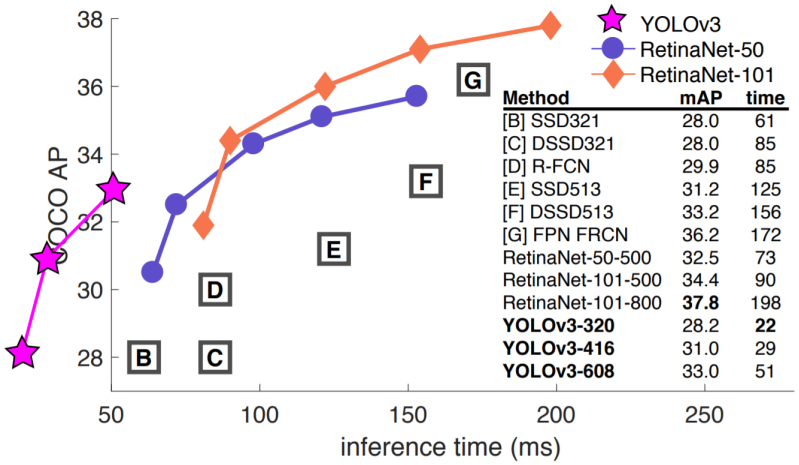

Figure 1.4: Comparison between YOLO v3 and other algorithms in terms of inference time.

As the above graph highlights, YOLO v3 runs much faster than other detection methods with a comparable performance using an M40/Titan X GPU.

1.2 Network blocks

In order to re-implement YOLO v3 in PyTorch, first of all, we analyzed Darknet's `yolov3.cfg` configuration file to look out for patterns. The original network configuration has five different types of blocks, of which:

- **convolutional** blocks which stand for plain convolutional layers to be added to the network (with some given hyperparameters), followed by a batch normalization layer and a Leaky ReLU activation function;
- **shortcut** blocks performing residual connections;
- **yolo** blocks doing nothing more than transforming the input: indeed, these are the blocks devoted to computing the loss value at the three output scales;
- **route** blocks which either return the output of some previous layer (this scenario happens after each **yolo** block) or perform a concatenation of outputs of two previous layers;
- **upsample** blocks that stand for plain upsample layers to be added to the network.

A deeper analysis of the network configuration allows to extract an easier representation of it and therefore an easier customization.
Our developed network configuration has the following blocks:

- **convolutional**, which are equal to the convolutional blocks of the previous configuration;
- **residual**, which are a combination of two consecutive **convolutional** blocks followed by a residual connection (i.e., the previous **shortcut** block), repeated as many times as required by the block's hyperparameters;

- **scale prediction**, which are a combination of one **convolutional** *block* and one **convolutional** block without the batch normalization and Leaky ReLU (i.e., a plain convolutional layer) in which the latter outputs a tensor with all the data for computing the loss and performing the object detection;
- **detection**, which are a combination of three **convolutional** blocks and a **scale prediction** block. This last block is excluded from the forward pass in order not to need, as for the previous network configuration, a **route** block returning the output of its predecessor.

1.2.1 Convolutional

As we briefly described earlier, the **convolutional** block represents a concatenation of three layers:

- a **two-dimensional (2D) convolutional** layer (two dimensions since we are dealing with images);
- a **batch normalization** layer;
- a **Leaky ReLU** as activation function.

We shall regard these layers as the function

$$ConvBlock(x) \overset{def}{=} LeakyReLU(BatchNorm(Conv2d(x))).$$

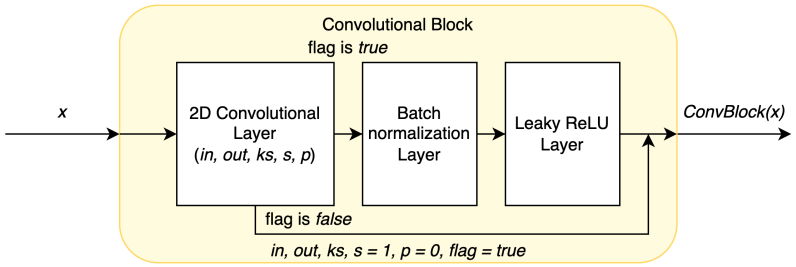

Figure 1.5: Convolutional block: graphical representation

For this block we require the following hyperparameters:

- the **number of input channels** *in* which, in the case of the first block of the network, is the number of color channels of the input image;
- the **number of output channels** *out* which generally corresponds to the number of convolutional kernels to be learned and to the number of values to be predicted when they are found last inside a **scale prediction** block;
- the **convolutional kernel size** ks representing a kernel of receptive field $ks \times ks$, hence dimensions $ks \times ks \times in$;
- the **padding** p, with a default value of 0, to apply both to the width and to the height;

- the **stride** s, with a default value of 1;
- a boolean flag, *flag*, with a default value of `true`, which controls whether to add a batch normalization layer and a Leaky ReLU activation function after the 2D convolutional layer (of course, when the flag is `false` then $ConvBlock(x) \overset{def}{=} Conv2d(x)$).
  In order to distinguish between the two, sometimes we will use the notations $ConvBlock_{batch}(x)$ and $ConvBlock_{nobatch}(x)$.

Given the input x, the forward pass returns $ConvBlock(x)$.

1.2.2 Residual

A **residual** block, as we briefly described earlier, is nothing more than two **convolutional** blocks $ConvBlock_1$, $ConvBlock_2$ with a residual connection, repeated a fixed amount of times n.
One important detail about this block is that the shape of the output tensor from the two **convolutional** blocks is kept equal to that of the input, in order to sum residuals component-wise. This is done by the first block having the output channels $out_1 = \frac{in_1}{2}$, a stride $s_1 = 1$, a padding $p_1 = 0$, and a kernel size $ks_1 = 1$; the second block having $in_2 = out_1$, $s_2 = 1$, $p_2 = 1$ and $ks_2 = 3$.
Let's regard this block as the function

$$ResBlock_n(x) \overset{def}{=} \begin{cases} ResBlock_{n-1}(x) + ConvBlock_{n,2}(ConvBlock_{n,1}(ResBlock_{n-1}(x))), & n > 1 \\ x & n = 1 \end{cases}$$

with n the number of repetitions, yielding a total amount of $2n$ **convolutional blocks** and n residual connections.

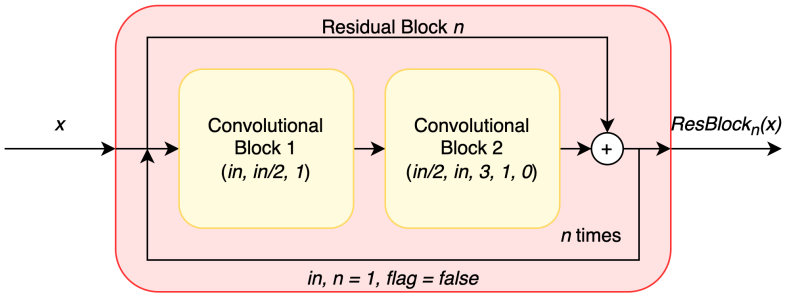

Figure 1.6: Residual block: graphical representation

The hyperparameters for this block are:

- the **number of input channels** in ($= in_1$), which needs to match the number of output channels of the previous block;
- the **number of repetitions** n (default value of 1), which is the number of times the couple of **convolutional** blocks needs to be added to the block list of the network;
- a boolean flag, *flag*, with a default value of `false`, for saving the output of the block to perform a route connection (which is what was done by the **route** block in the previous configuration).

Given the input x, the forward pass returns $ResBlock_n(x)$ and if the flag is set to `true`, the returned value is stored by the network for later use.

1.2.3 Scale prediction

This is a special block because its output does not contribute to the network itself but only to the loss function value, which is explained in Section 3.4. In the whole network there are just three of these blocks and each one contributes to one third of the loss, namely ℓ_i, with $i \in \{1, 2, 3\}$.
It is composed by a first **convolutional** block $ConvBlock_{batch}$ with $out_{batch} = 2in_{batch}$, $s_{batch} = 1$, $p_{batch} = 1$, $ks_{batch} = 3$ and a second **convolutional** block $ConvBlock_{nobatch}$ with $out_{batch} = (5 + \#classes) \cdot \#anchors_i$, $s_{nobatch} = 1$, $p_{nobatch} = 0$, $ks_{nobatch} = 1$, without the batch normalization and Leaky ReLU.
For the ImageNet dataset, for instance, $\#classes = 1000$ and $\#anchors_i = 3 \ \forall i \in \{1, 2, 3\}$.
This block can be represented by the function

$$ScalePredBlock(x) \overset{def}{=} ConvBlock_{nobatch}(ConvBlock_{batch}(x)).$$

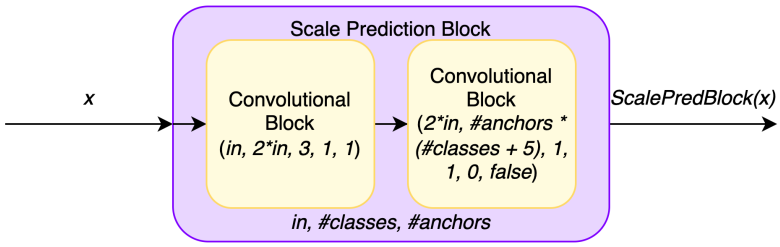

Figure 1.7: Scale Prediction block: graphical representation

The hyperparameters are:

- the **number of input channels** in ($= in_{batch}$), which needs to match the number of output channels of the previous block;
- the **number of classes** $\#classes$ of the dataset which need to be predicted;
- the **number of anchors** $\#anchors$ of the dataset for the scale level i which the block predicts the loss for.

Please note that even if our implementation allows to set $\#anchors$ for each scale, we decided to stick with the original idea of having three anchors per **scale prediction** block as in the original implementation ($\#anchors_i = 3 \ \forall i \in \{1, 2, 3\}$, as we said earlier).
Given the input x, the forward pass outputs y'', which is:

$$y'' = (y'_1 \ y'_2 \ y'_4 \ y'_5 \ y'_3) \text{ with } y'_i \text{ the } i\text{-th column of } y',$$

$$y' = reshape_{(batch\_size, \ \#anchors, \ 5 + \#classes, \ grid\_size, \ grid\_size)}(ScalePredBlock(x)),$$

considering x having an original shape of $batch\_size \times \ldots \times grid\_size \times grid\_size$.

1.2.4 Upsample

This block is merely a wrapper around an upsample layer, with **bilinear** as the upsampling algorithm. This block can be represented with the function

$$UpsampleBlock(x) \overset{def}{=} Upsample(x).$$

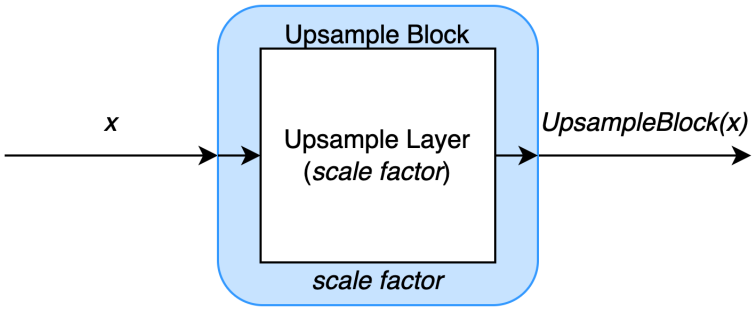

Figure 1.8: Upsample block: graphical representation

There is just one hyperparameter, which is the **scale factor**, namely the factor used for increasing the last axis dimensions of the input tensor.
Given the input x, the forward pass outputs $UpsampleBlock(x)$ and the network updates the current output tensor with the concatenation of $UpsampleBlock(x)$ and the output of the last $ResBlock_n(x)$ with the flag set to `true`.

1.2.5 Detection

This last block we are describing performs the final steps before having the partial loss ℓ_i, with $i \in \{1, 2, 3\}$, as the result of the **scale prediction** block.
It is made of three **convolutional blocks**, the first two acting exactly like those in the **residual** block but without the summation of x; the third performing 1×1 convolutions, preparing the input for the **scale prediction** block.
A function representing this block can be

$$DetectionBlock(x) \overset{def}{=} ConvBlock_3(ConvBlock_2(ConvBlock_1(x))).$$

The hyperparameters are the following:

- the **number of input channels** *in*, which needs to match the number of output channels of the previous block;
- the **number of output channels** *out*, that is the number of convolutional kernels to be learned and the input channels for the **scale prediction** block;

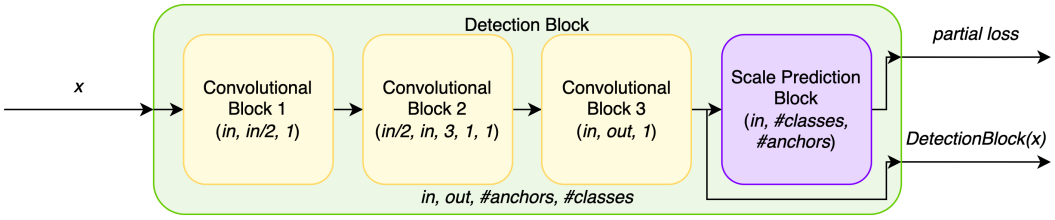

Figure 1.9: Detection block: graphical representation

- the **number of classes** $\#classes$ of the dataset which need to be predicted (forwarded to the **scale prediction** block);
- the **number of anchors** $\#anchors$ of the dataset for the scale the block predicts the loss for (forwarded to the **scale prediction** block).

Given input x, the forward pass outputs $y = DetectionBlock(x)$ and the network computes the partial loss $\ell_i = ScalePredBlock(y)$.

1.3 Network layout

With the blocks defined in the previous section, we present the base architecture of the YOLO v3 network.

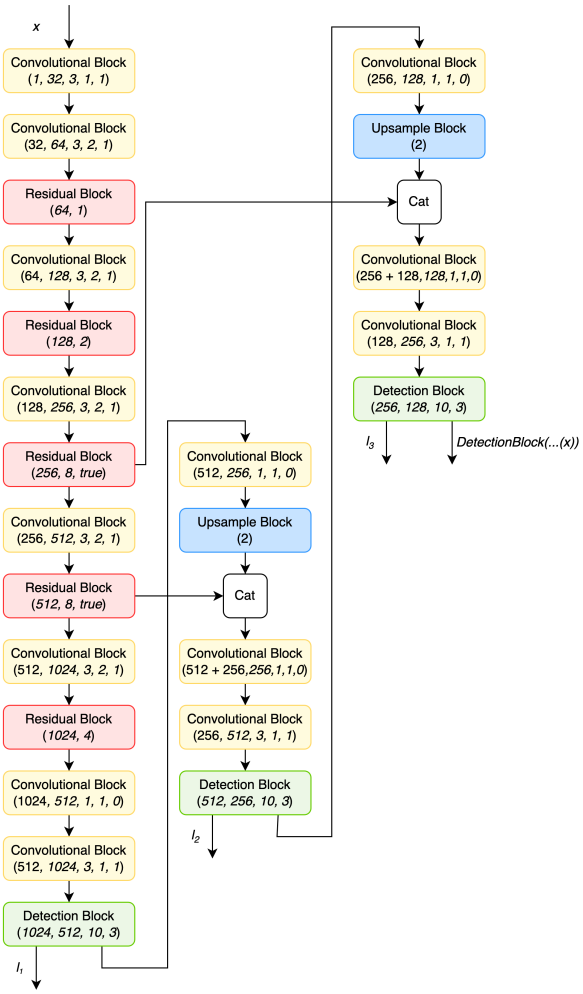

Figure 1.10: YOLO v3 base network architecture: graphical representation

This architecture respects the specification of the original YOLO v3 architecture. As we will describe in the following section, some adjustments were made to improve the convergence speed. In the diagram we clearly notice what we described in the previous sections:

- first of all, we notice that at the end of each of three "column of blocks" there is a **detection block** which returns the output tensor $DetectionBlock(\dots(x))$ and ℓ_i;
- there are two **residual** blocks which have the boolean flag set to `true` and indeed their value is stored inside the network, as it is shown, to be concatenated with the output of the two **upsample** blocks;

- the downsampling, as we briefly described in Section 3.1.1 and Section 3.1.2, which takes an input image of size $w \times h$ and reduces it by factors 2^5, 2^4, 2^3 (these are obtained by multiplying the stride of all the **convolutional** blocks until each **detection** block) to obtain three **feature maps** at three different recognition scales (which discover progressively smaller objects). With our dataset, it means that the input images of size 128×128 gets transformed to three feature maps of sizes 4×4, 8×8, 16×16.

1.4 Loss function

The loss function of the network is quite complex since it needs to consider many aspects:

- the bounding boxes, which are described by their width w, height h, center coordinates (x, y);
- the objectness score, which describes how much a given bounding box is likely to be recognizing an object;
- the no-objectness score, which describes how much a given bounding box is likely *not* to be recognizing an object;
- the class probabilities, which describe how likely it is that the bounding box is recognizing an object of each class.

Moreover, the final loss value ℓ is obtained by summing the three partial losses, hence $\ell = \sum_{i=1}^{3} \ell_i$.
The output of *ScalePredBlock*, at each scale i, is a tensor $\hat{out}$ of shape ($batch\_size$, $\#anchors_i$, $grid\_size_i$, $grid\_size_i$, 5+$\#classes$), while the corresponding target out has shape ($batch\_size$, $\#anchors_i$, $grid\_size_i$, $grid\_size_i$, $5+1$).
Please note that, for the sake of simplicity, we will not consider the $batch\_size$ in the loss formula (we shall regard it as $batch\_size = 1$). It would only add further summation and additional subscript index. Finally, the loss function at this scale is the following:

$$\ell_i = \lambda_{noobj} \cdot \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} \mathbb{1}_{k,i,j}^{noobj} \frac{LogLoss(\sigma(\hat{c}_{k,i,j}), c_{k,i,j}))}{\#noobjects} \quad \text{(no object loss)}$$

$$+ \lambda_{obj} \cdot \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} \mathbb{1}_{k,i,j}^{obj} \frac{SquaredError(\sigma(\hat{c}_{k,i,j}), IoU(obj\_\hat{bbox}_{k,i,j}, obj\_bbox_{k,i,j}))}{\#objects} \quad \text{(object loss)}$$

$$+ \lambda_{box} \cdot \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} \mathbb{1}_{k,i,j}^{obj} \frac{SquaredError(\hat{bbox}_{k,i,j}, bbox_{k,i,j})}{\#objects} \quad \text{(box loss)}$$

$$+ \lambda_{class} \cdot \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} \mathbb{1}_{k,i,j}^{obj} \frac{LogSoftmax_C(\hat{p}_{k,i,j}, p_{k,i,j})}{\#objects} \quad \text{(class loss)}$$

with:

- $A = \#anchors_i$
- $S = grid\_size_i$
- $C = \#classes$

- $anchor_i = [anchor\_w \;\; anchor\_h]$
- $\forall k \in [1, A],\ i, j \in [1, S]\ \mathbb{1}_{k,i,j} = \begin{cases} 1 & \text{bbox contains an object and has the highest IoU at scale } k \\ -1 & \text{bbox contains an object but does not have the highest IoU at scale } k \\ 0 & \text{otherwise} \end{cases}$
- $\#objects = \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} (\mathbb{1}_{k,i,j}^{obj} = 1)$[3]
- $\#noobjects = \sum_{k=1}^{A} \sum_{i=1}^{S} \sum_{j=1}^{S} (\mathbb{1}_{k,i,j}^{obj} = 0)$[3]
- $\forall k \in [1, A],\ i, j \in [1, S]\ \hat{out}_{k,i,j} = [\hat{c} \;\; \hat{x} \;\; \hat{y} \;\; \hat{w} \;\; \hat{h} \;\; \hat{p}],\ \hat{p} = (p_1 \;\; \dots \;\; p_C)$
- $\forall k \in [1, A],\ i, j \in [1, S]\ \ out_{k,i,j} = [c \;\; x \;\; y \;\; w \;\; h \;\; p]$
- $LogLoss(x, y) = y \cdot log\, x + (1 - y) \cdot log(1 - x)$
- $SquaredError(x, y) = (x - y)^2$
- $LogSoftmax_C(x, y) = log\, \frac{exp(x_y)}{\sum_{c=1}^{C} exp(x_{classes_C})}$
- $\hat{obj\_bbox}_{k,i,j} = [\sigma(\hat{x}_{k,i,j}) \;\; \sigma(\hat{y}_{k,i,j}) \;\; exp(\hat{w}_{k,i,j}) \cdot anchor\_w_k \;\; exp(\hat{h}_{k,i,j}) \cdot anchor\_h_k]$
- $obj\_bbox_{k,i,j} = [x_{k,i,j} \;\; y_{k,i,j} \;\; w_{k,i,j} \;\; h_{k,i,j}]$
- $\hat{bbox}_{k,i,j} = [\sigma(\hat{x}_{k,i,j}) \;\; \sigma(\hat{y}_{k,i,j}) \;\; \hat{w}_{k,i,j} \;\; \hat{h}_{k,i,j}]$
- $bbox_{k,i,j} = [x_{k,i,j} \;\; y_{k,i,j} \;\; log\, \frac{w_{k,i,j}}{anchor\_w_k} \;\; log\, \frac{h_{k,i,j}}{anchor\_h_k}]$.

1.5 Network training

1.5.1 Dataset

Regarding datasets, we create the `DataLoader` objects for training, evaluation and testing. Moreover, we make use of the *Automatic mixed precision package* offered by "PyTorch": it allows to perform some operations by using the `torch.float32` (standard single precision floating point number) datatype and other operations by using `torch.float16` (half precision floating point number).
Some operations, like linear layers and convolutions, are much faster in `float16`, while other ones, like reductions, often require the dynamic range of `float32`: mixed precision tries to match each operation to its appropriate datatype.

[3] This is a little abuse of notation: we mean "return 1 when $\mathbb{1}_{k,i,j}^{obj}$ equals 1 (or 0), 0 otherwise".

1.5.2 Early stopping

Our training process implements an **early stopping** mechanism: in the configuration file we set the tolerance regarding the number of epochs after which the specified metric does not improve. Our specified metric is the mean loss value averaged over the batches needed to complete an epoch, namely the mean loss value during that epoch.
We actually begin checking the evolution of the mean loss after the 5th epoch in order not to stop too early those training starting from a bad point in the loss function space.

1.5.3 Performance measures

Regarding performance measures, we track the evolution of the *class prediction accuracy*, the *object accuracy*, the *no object accuracy*, the *mean loss* and the *Mean Average Precision (mAP)*, over all the epochs: in detail, we plot on the same graph the evolution of every aforementioned metric during the training and the validation, so that they are able to highlight potential underfitting or overfitting.

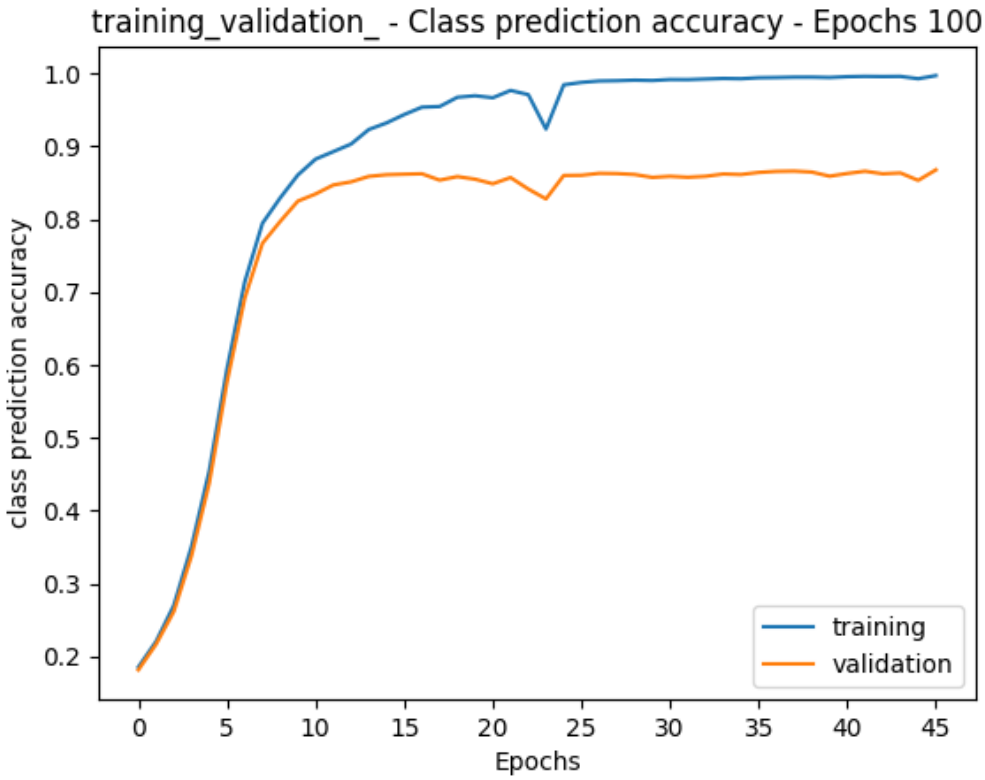

Figure 1.11: Class prediction accuracy during training and validation steps.

Appendix A

References

A.1 Papers

1. J. Redmon, S. Divvala, R. Girshick, A. Farhadi; You Only Look Once: Unified, Real-Time Object Detection; 2016
2. J. Redmon, A. Farhadi; YOLOv3: An Incremental Improvement; 2018

A.2 Articles

1. YOLOv3 theory explained
2. What's new in YOLO v3?
3. Calculating Loss of Yolo (v3) Layer
4. YOLO V3 Explained
5. Tech notes of implementation of YOLO V3
6. Yolo v3 loss function
7. Implementation of YOLOv3 on Malaria Cells Dataset
8. Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3
9. How to implement a YOLO (v3) object detector from scratch in PyTorch
10. YOLOv3 — Implementation with Training setup from Scratch
11. YOLOv3: An Incremental Improvement
12. YOLOv3: Real-Time Object Detection Algorithm

A.3 Other material

1. A. Asperti; Slides of the course "Machine Learning"; 2021