UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MIARFID Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Department of Computer Systems and Computation
Universitat Politècnica de València

Detection of Mathematical Expressions in Scientific Papers

MASTER'S THESIS

Master's Degree in Artificial Intelligence, Pattern Recognition and Digital Imaging

Author: Salvador Carrión Ponz

Tutor: Roberto Paredes Palacios

Course 2018-2019

Abstract

The field of object detection drew the attention of the scientific community in 2013 when R-CNN[1] outperformed by a 30% margin the previous best results on the PASCAL VOC 2012 challenge[2]. Since then, researchers all over the world have designed new models that significantly outperform R-CNN[1], both in terms of accuracy and speed. Among these new models, YOLO[3] stood out for its extremely fast detection rate and accuracy; and SSD[4] for the novel discretization of the output space and multi-scale prediction.

In this work, we address the problem of detecting mathematical expressions in scientific papers. First, a brief introduction to the problem of general-object detection is given, along with state-of-the-art techniques and evaluation methods. Then, the two aforementioned state-of-the-art models (YOLO[3] and SSD[4]) are presented, and a comparative study is carried out where we analyze the problems and weaknesses found when they are applied to this specific task. Finally, the work ends with a discussion on how to tackle these problems in future models and the possibility of deriving this work towards the mathematical expression recognition problem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Humans are incredibly good at performing object detection tasks. For instance, we can take a look at an image and instantly know what objects are in there, where to find them, and even, we can predict the location of objects that cannot be seen but there is a hint that they are hidden somewhere.

In 2013, a team of neuroscientists from MIT found that the human brain can process images in 13 milliseconds[5]. This means that to see, our brain has to processes a huge amount of information using extremely efficient biological mechanisms developed throughout evolution. The problem is, we don't know how we do it. We just do it. To overcome this problem, researchers have struggled for many years to mathematically define what does it mean to *see*, or at least, for the eyes of a computer. Thankfully for us, nowadays we have some tricks to make computers able to *see* (metaphorically speaking).

In this work we wanted to use some of these *tricks* (from now and on, we will call them *models*), to detect mathematical expressions in scientific papers. Inspired by the efficiency of the human brain and the constraints of a real-world scenario, we have focused our research on single-shot models.

1.1 Motivation

Object detection has woken up a great interest in recent years among researchers and companies due to its relevance for many industries and interesting applications: self-driving cars, drones, surveillance systems, medical imaging, media analysis, sports, manufacturing, automatic counting, etc.

Among the many problems that we could tackle in this field, we noticed that mathematical expression detection is not a popular one and it could very interesting for the Optical Character Recognition (OCR) industry.

1.2 Related work

Before the popularity of convolutional neural networks (CNN) the two most popular approaches for object detection were Deformable Part Models (DPM)[6] and Selective Search[7]. These methods usually worked with either sliding windows or region proposals. But when R-CNN[1] came to scene combining Selective Search[7] with CNNs, region proposal methods took the lead and became the usual approach.

From R-CNN[1], many new models have emerged that improve their original results, both in terms of accuracy and speed.

- Multibox[8]: Saliency-inspired neural network model for detection. It predicts a set of class-agnostic bounding boxes along with their likelihood of containing an object of interest. Unlike the rest of the models here presented, Multibox only locates saliency objects in an image (it doesn't perform classification).
- R-CNN[1]: Uses Selective Search[7] for the region proposals and a CNN to perform classification over the proposed regions. Although this model significantly improved the accuracy of previous detectors, it was too slow since each region proposal has to go through the CNN.
- Overfeat[9]: Uses a CNN to classify multiple locations in the image, in a sliding window fashion, and over multiple scales.
- SPP-Net[10]: Introduces spatial pyramid pooling to remove the fixed input size requirement in most of the previous CNN-based detectors. Faster than R-CNN[1] with better or comparable accuracy. Uses a multi-stage pipeline as R-CNN[1] does.
- Fast R-CNN[11]: Inspired by SPP-Net[10], Fast R-CNN[11] improved R-CNN[1] by sharing the computation of convolutional layers between regions proposals using what they called, a *RoI Pooling* layer, which essentially is a special case of Spatial Pyramid Pooling (SPP) layer with just one pyramid level.
- Faster R-CNN[12]: Extends Fast R-CNN[11] to share the full-image convolutional features with the detection network. This releases it from using (the expensive) Selective Search[7] to generate the region proposals. Instead, it introduces an extra fully convolutional network, called Region Proposal Network (RPN).
- YOLOv1[3]: Introduces an end-to-end extremely fast neural network to predict bounding boxes and class probabilities directly from full images in one evaluation.
- SSD[4]: A single network that eliminates the proposal generation and subsequent pixel or feature resampling stages. Additionally, it performs multi-scale predictions by combining feature maps at different resolutions.
- YOLOv2[13]: Improves YOLOv1[3] using a better but lighter network, batch normalization (instead of dropout[14]), high-resolution classifier, anchor boxes, dimensional clusters, direct location prediction, fine-grained features, and multi-scale training.
- YOLOv3[15]: Incremental improvement over YOLOv2[13]: i) Objectness score for each bounding box using logistic regression; ii) Independent logistic classifiers for the class predictions; iii) Predictions across three scales; iv) Bigger feature extractor with similar speed but better accuracy.

1.3 Overview of the proposal approach

First, we start with a comparative study of two well-known object detection models: YOLOv3[15] and SSD[4]. For the study, we will try to run the experiments under the same conditions as long as these don't harm the results of a model in favor of the other. That is to say, we will force both models to use the same data augmentation, similar training time/iterations, same hardware, similar hyper-parameters, same evaluation metrics, etc.

After this experiment, we will analyze the weaknesses of each model when dealing with the specific tasks of mathematical expression detection. The analysis of these weaknesses will be split into two parts: i) first, a subjective assessment on how well the models are at detection equations; ii) and two, an error analysis following objective metrics such as recall, precision, mAP,... For the latter part, we could use a similar methodology as proposed by Hoiem et al.[16].

Then, we will discuss implementation details and problems that we had to deal with during the study.

As these proposed models are designed for a similar object detection task but with different requirements, we likely need to perform a bit of *model surgery* to make them work properly. Depending on the degree of these modifications, the final models might diverge a bit from the ones published in their original papers. For instance, we could perform modifications in the models such as changing the base feature extractor, architecture modifications to reduce the number of region proposals (we're dealing with big images), additions of extra multi-scale layers, simplifications to increase the speed, etc.

During the whole study, we will use 90% of the data for training and the remaining 10% for testing.

1.4 Context of applications and assumptions

The direct application of this work is to improve Optical Character Recognition (OCR) systems. Usually, these systems are made up of many subsystems to detect specific things that depend on the application such as title/subtitles/body in a book, plate licenses in a parking lot, figures or tables in a paper, etc. One of these specialized systems could be a module to retrieve all the mathematical expressions in a book as a text format.

This is not an easy problem, but in order to feed the highly complex and specialized mathematical expression recognition module, we first need a detection module that allows us to locate the bounding boxes of the mathematical expressions found in a given image.

Another application derived from this ORC system could be an app that solves math equations without the need for the user to write them, or we could even derive a metric for an online bookstore to know the mathematical level required to read a book or paper. Possibilities are countless!

1.5 Expected outcomes/results

YOLO[15] has come a long way since the publication of its first version in 2015[3]. Until now, three versions of YOLO exists.

On the other hand, SSD[4] was published a few months later than YOLOv1[3]. And up to my knowledge, there are no more published versions of it.

Science is built upon previous works so it is expected that the follow-up versions of YOLO include ideas from better models. For YOLOv2[13], one of these inspirational models was SSD[4]. And two years later, YOLOv3[15] superseded YOLOv2[13] with an incremental improvement.

For this reason, it is expected that YOLOv3[15] presents the best results among these two models. Nevertheless, I do expect significant improvements in both models if state-of-the-art feature extractors (aka classifiers) are used.

Chapter 2

Object detection with deep learning

2.1 SSD: Single Shot Detector

SSD[4] is a single-shot object detector that appeared after YOLOv1[3]. It is a purely convolutional neural network and can be conceptually organized into three parts:

- **Feature extractor**: Base model derived from a classification architecture to provide low-level feature maps.
- **Auxiliary layers**: Convolutions appended to the end of the feature extractor to provide high-level feature maps.
- **Prediction layers**: Convolutions used to locate and classify objects in the feature maps.

2.1.1 Feature extractor

For the base feature extractor, it is common to use a truncated version of a good classification network such as VGG[17] or ResNet[18]. Why? Because models that had been proven to work well in image classification tasks are good at capturing the semantics of an image.

In object detection, we are not that interested in the image as a whole but in the specific regions where the objects are located. Luckily for us, the same convolutional features that are useful for object classification are useful for object detection.

Furthermore, there are more advantages form using reliable classification models. One of them is the possibility of using a pre-trained model to perform Transfer Learning. This means that we can use the knowledge from a different but closely related task and transfer it to our model; usually shorting the training times and improving the initial accuracy, although there might be great benefits from training from scratch. As usual, it depends on the specific task.

Equally important, to adapt the model to our domain some changes in the original architecture had to be made (see Figure 2.1). The changes we did, were:

- Use a bigger input image
- Introduce a ceiling operation to avoid inconvenient feature sizes (related to rounding).

- Rework fully connected layers into fully convolutional.

- Modify convolutional layers to get the desired feature maps. To do this, we could play with the stride, the padding, the pooling, the subsampling, etc.

To perform the reduction needed to decrease the size of the features maps, we used a technique known as *decimation*, where each n-th element is we subsampled along a particular dimension. In convolutional layers, this can be easily achieved through the use of a kernel with a dilation greater than one.

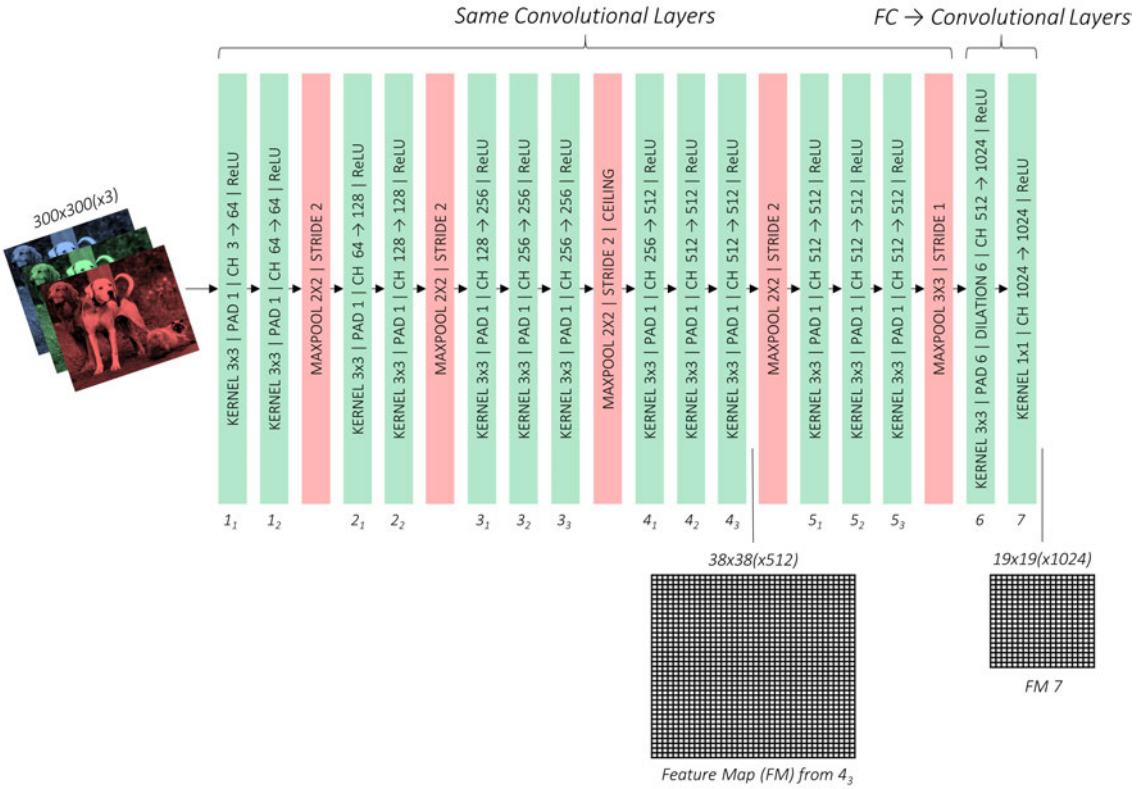


Figure 2.1: **Feature extractor**: SSD uses a modified VGG16 network to understand the semantics of the image [19]

2.1.2 Auxiliary layers

Auxiliary layers are simply a series of convolutional layers on top of the base feature extractor to provide additional feature maps.

To allow multi-scale detection, these convolutional layers are progressively decreased in size (see Figure 2.2). Bigger feature maps can easily capture small objects, while smaller feature maps capture big objects.

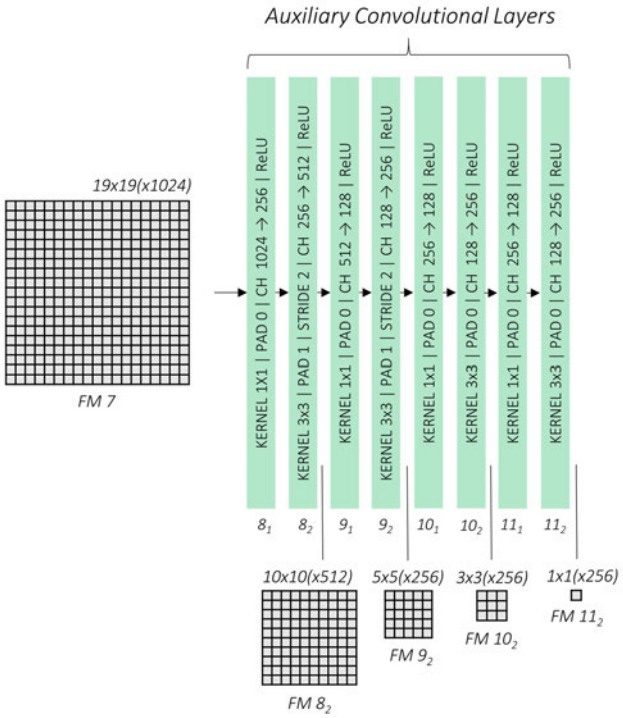


Figure 2.2: **SSD auxiliary layers** [19]

2.1.3 Prediction layers

A problem of having predictions encoded in the feature maps is that it is not intuitive how to transform them into something that we can easily understand.

We want to predict:

- Offset coordinates, for each bounding box.
- N scores corresponding to all the class probabilities and the background, for each bounding box.

To do achieve this, we add two additional convolutional layers for each prediction feature maps: (see Figure 2.3)

- A *location convolutional-layer* evaluating at each location of the feature map, with a kernel of $3x3$ and 4 filters per prior.
- A *classification convolutional-layer* evaluating at each location of the feature map, with a kernel of $3x3$ and *n_class* filters per prior.

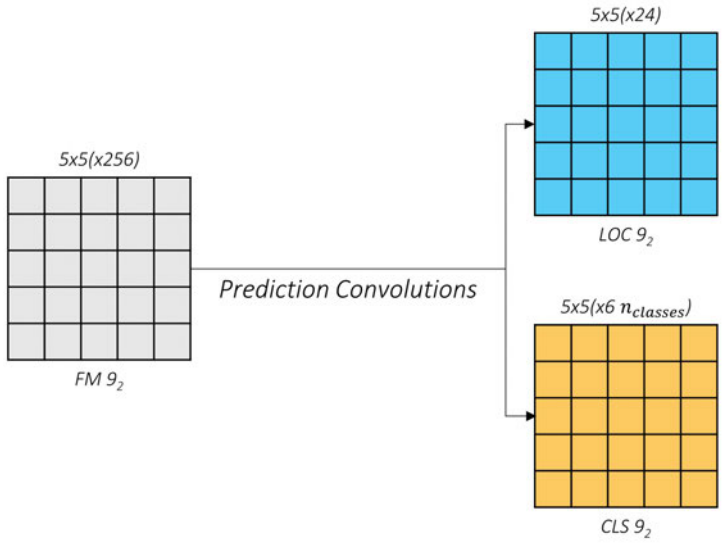


Figure 2.3: **Prediction layers of SSD**: Localization and classification [19]

It is important to remark that the actual predictions happen along the *depth* (see Figure 2.4). For instance, if a feature map has a size of $(5, 5)$, there are 6 priors and we want to predict the location coordinates, the *location feature map* will have a final dimension of $(5, 5, 24)$. The last channel means that for each location $(i, j, 24)$ there are 6 predictions with 4 offsets each. These offsets correspond to the coordinates of the predicted bounding boxes.

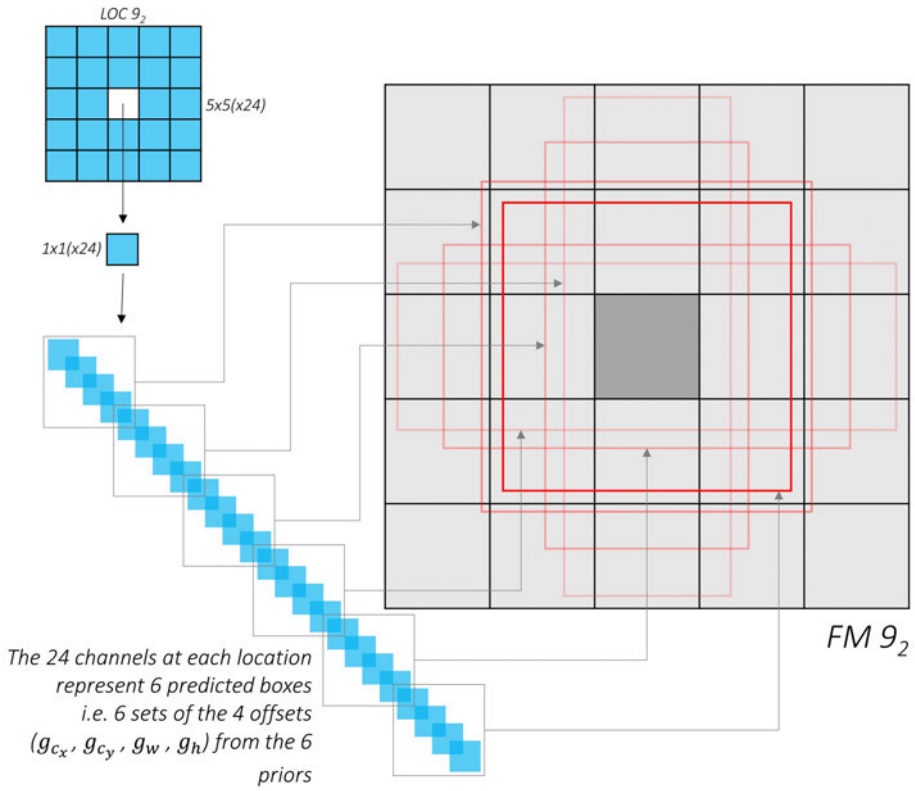


Figure 2.4: **Decoding predictions**: SSD encodes its predictions along the depth of its feature maps [19]

Finally, all the predictions coming from feature maps at different scales are stacked into a single tensor. This final tensor is the model's output.

2.1.4 Loss

Due to the intricacies of this model, SSD[4] introduces a unique loss function. This loss is an aggregation of a regression loss for the bounding boxes and a classification loss for the object type.

To compute this loss, we need to match the predicted bounding boxes to their corresponding ground truth. This is tricky since object detection is an open-ended task and predictions doesn't correspond to any specific ground truth. To match them we will compute their overlapping ratio (using the Jaccard Index) against all the ground truths objects.

The whole process is done as follows:

1. Find the Jaccard Index between all the priors and ground truth objects.
2. Match each Prior to the ground truth that has the greatest overlap.
3. Each prior is linked to a prediction, which is transitively matched with the ground truth assigned to its prior.

If the best overlap between a prior and all the ground-truths is greater than a threshold (usually 0.5), then that prior will be considered as *matched* to that ground truth object. Otherwise, it will be matched to the *background* class.

As it has been said before, the overall loss is a Multibox loss[8], an aggregation of the confidence loss and the localization loss:

$$L = L_{conf} + \alpha L_{loc} \tag{2.1}$$

Specifically, the exact loss is:

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \tag{2.2}$$

where:

- N: Number of positively matched prior boxes ($if N == 0 : L = 0$)
- L_{conf}: Confidence loss
- L_{loc}: Localization loss
- α: Parameter to control the relevance of the localization loss (usually, $\alpha = 1.0$)

The confidence loss is a measure of how well the model distinguishes the background from the relevant objects. As every prediction must have a ground truth class attached to it, we use a sort of classification loss where the confidence loss is simply the sum of the cross-entropy losses of the positive and the negative matches.

$$L_{conf} = \frac{1}{N_{pos}} \left(\sum_{i \in Pos} CE + \sum_{i \in Neg} CE \right) \tag{2.3}$$

To be rigorous with the loss presented in the original paper, here is the full version:

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{i,j}^{p} \log \hat{c}_i^p - \sum_{i\in Neg} \log \hat{c}_i^0 \tag{2.4}$$

where $x_{i,j}^{p} = \{0,1\}$ indicates the match for the i-th default box with the j-th ground-truth box of category p, and the $\hat{c}_i^p$ is the softmax loss of the classes confidences c_i:

$$\hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p exp(c_i^p)} \tag{2.5}$$

The localization loss is computed regressing only the positively matched predicted boxes. Why? Because the negative matches correspond to the background class, and there no need to regress bounding boxes for the background.

Consequently, the localization loss is the average *Smooth L1* loss of the positive matches:

$$L_{loc} = \frac{1}{N_{pos}} \left(\sum_{i\in Pos} Smooth_{L1} \right) \tag{2.6}$$

Or to be more technically consistent:

$$L_{loc}(x,l,g)) = \sum_{i\in Pos}^{N} \sum_{m\in\{c_x,c_y,w,h\}} x_{i,j}^{k} \cdot smooth_{L1}(l_i^m - \hat{g}_j^m) \tag{2.7}$$

where:

- l: Predicted box
- g: ground truth box
- c_x, c_y: Center offsets for the bounding box
- w, h: Width and Height of the bounding box
- α: Weight term (set to 1 by cross-validation)

The Smooth L1 loss is used because it is less sensitive to outliers than the Mean Squared Error (MSR) and in some cases, it might prevent the exploding gradients phenomenon[11]. To achieve this, it uses a squared term when the absolute element-wise error falls below 1 and an L1 term otherwise:

$$Smooth_{L1}(x,y) = \sum_i z_i \tag{2.8}$$

where z_i is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & if \quad |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & otherwise \end{cases} \tag{2.9}$$

2.1.5 Priors

Generally speaking, object detections can occur in any place, position, size, aspect ratio,... There so many possibilities for each object that they can't be enumerated. But clearly, there are some possibilities more likely than others. For instance, bounding boxes on people tend to be taller than wider, for cars the opposite, and birds are found in the sky more often than dogs. Hence, we could try to collectively represent the universe of probable bounding boxes by using *priors*, that is, boxes that approximate this object-space universe.

The authors chose the priors manually, following a careful method based on a grid of feature maps, scales and aspect ratios.

For example, a feature map of size (10, 10) with a prior scale of 0.5 and these aspect ratios [1:1, 2:1, 1:2, 3:1, 1:3] (5 positions), will result in a total of 600 priors[1].

In the paper, the prior scales are equally spaced between feature maps:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), \quad k \in [1, m] \tag{2.10}$$

where:

- s_{min}: Minimum scale (here, $s_{min} = 0.2$)
- s_{max}: Maximum scale (here, $s_{min} = 0.9$)
- m: Number of feature maps

2.1.6 Hard Negative Mining

A typical problem in object detection is that the vast majority of the predictions made by the model do not contain any object.

If the number of negative matches strongly overwhelm the positive ones, the model will take a *shortcut* and predict most objects as negative. To fix this, or better, to mitigate this effect, we have to apply some correction such as weighting the negative matches or limiting the number that will be evaluated in the loss function so that the imbalance can be corrected.

SSD[4] approaches this problem by applying *Hard Negative Mining*. That is to say, we take the negative predictions that the model was most wrong about and then, we feed the model again but using those wrong predictions in a fixed proportion with respect to the positive matches. The authors of SSD[4] decided to use three times more hard negatives than positives and considered the cross-entropy loss as the indicator of how hard is a specific match for the model.

2.2 YOLO: You Only Look Once

YOLO[3] is an extremely fast single-shot neural network to perform object detection. It *sees* object detection as a regression problem, predicting bounding boxes and class probabilities directly from images in one evaluation, outperforming many detectors both in terms of speed and accuracy.

[1] Total priors = $10 \cdot 10 \cdot 5 = 500$

One of the key aspects in YOLO is that instead of using a sliding window or a region proposal method, YOLO reasons globally about the image, encoding contextual information in its feature maps.

Since its original publication in 2015, two new versions of YOLO have been published. For this work, we have chosen YOLOv3[15], currently, the latest and most accurate version of the YOLO series.

Conceptually, YOLO divides the input image into an SxS grid, and if the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Then, each grid cell predicts B bounding boxes.

Each bounding box has associated 5 predictions plus C conditional class probabilities:

- x, y: Centered relative to the grid cell ($x \in [0..1]$)
- w, h: Width and height
- obj_{conf}: Confidence score (Objectness)
- $C_1, C_2, ...C_n$: Conditional class probabilities

The confidence score for each box can be defined as:

$$Obj_{conf} = Pr(Object) * IOU^{truth,prediction} \tag{2.11}$$

And the conditional class probability as:

$$C_i = Pr(Class_i|Object) \tag{2.12}$$

To know the class-specific confidence scores, we have to multiply the conditional class probability and the individual box confidence prediction (for each class):

$$Pr(Class_i) = Pr(Object) * Pr(Class_i|Object) \tag{2.13}$$

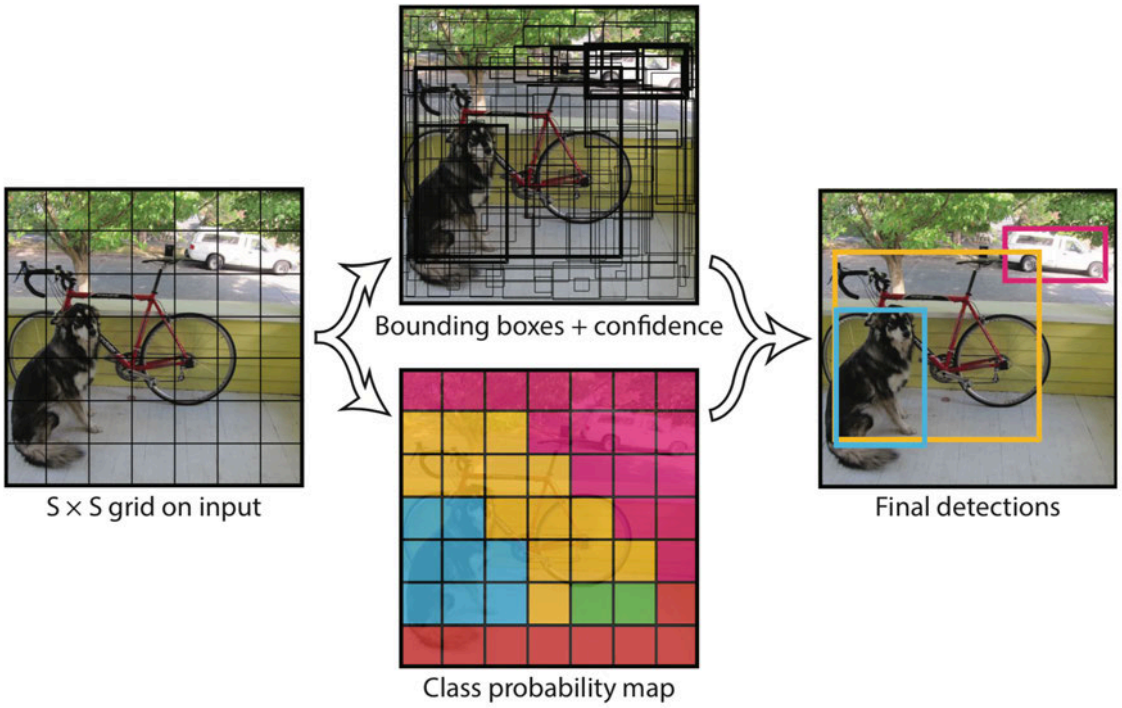


Figure 2.5: **Prediction concept in YOLO**: For each image, the model will output a tensor of fixed size equal to $SxSx(Bx5xC)$ [3]

From a network design perspective, YOLOv3[15] it can be organized into two parts:

- **Feature extractor**: Base model derived from a classification architecture for feature extraction.
- **Prediction layers**: Convolutions to perform predictions across different scales.

2.2.1 Feature extractor

Different versions of YOLO use different feature extractors.

- YOLOv1[3] introduces a GoogLeNet-inspired architecture, with 1x1 reduction layers followed by 3x3 convolutional layers[20], instead of the inception module[21].
- YOLOv2[13] uses Darknet-19, built off prior work on network design. It's made up of 19 convolutional layers and 5 max-pooling layers.
- YOLOv3[15] redesigns Darknet-19 into Darknet-53, a more accurate but slightly slower version of its predecessor.

Although Darknet-53 is an incremental improvement over Darknet-19, it's much more powerful than its predecessor and also more efficient than many of the current state-of-the-art classifiers. Furthermore, it uses less floating-point operations, which results in a substantial improvement of the network speed[15].

The architecture uses successive 3x3 and 1x1 convolutional layers but with some shortcut connections and few more layers (see Table 2.1).

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Table 2.1: **Darknet-53 architecture**

2.2.2 Prediction layers

YOLOv3[15] predicts bounding boxes across three different scales using a sort of feature pyramid networks[22].

At different points of the base feature extractor, several prediction layers are connected. The multi-scale predictions works as follows: i) The last prediction layer takes its input directly from end of the base feature extractor; ii) then, the second prediction layer takes two inputs, one from the end of the base network and another from the middle; iii) finally, the third prediction layer takes one input from the middle of the network and another from the second prediction layer. See Figure 2.6.

This pyramid of feature extraction approach allows YOLOv3[15] to detect objects across scales by extracting more meaningful semantic information from the upsampled features and finer-grained information from the earlier feature maps[15]. This is because feature maps from different levels within a network are known to have different receptive field sizes[23]. Namely, small feature maps are good at detecting big objects whereas big feature maps are good at detecting small objects.

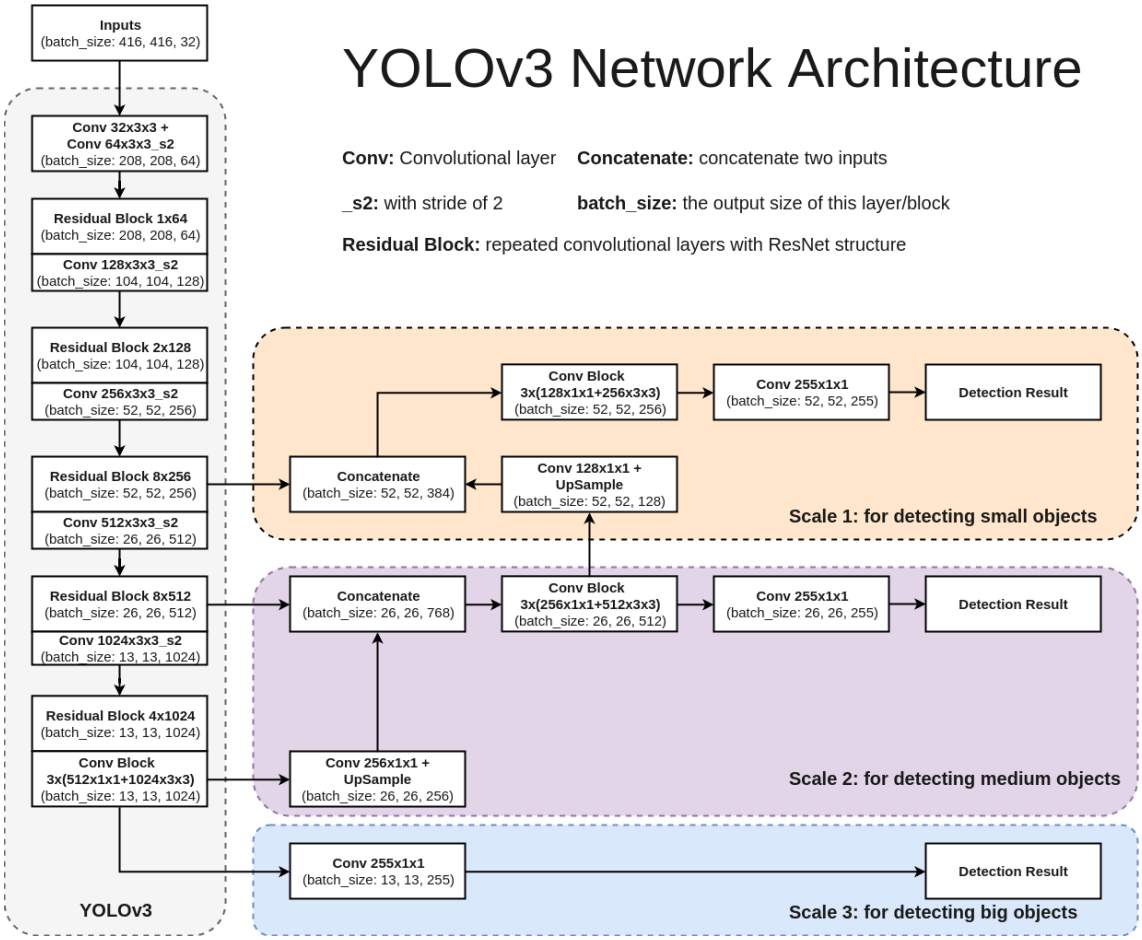


Figure 2.6: **Network architecture of YOLOv3** [24]

Each prediction layer outputs a set of bounding boxes as a feature map. To transform this feature map into something we can easily understand we have to reshape it into an equivalent tensor of dimensions: *(num_samples, num_anchors, num_classes + 5, grid_size, grid_size)*. Technically, the meaning of each dimension is given in this very first moment. When values are thrown into the loss function and the model is optimized using that order, the minimization of the loss will force those values to acquire the meaning we want.

Once we have made sense of the feature map and we have the predictions in the form we want, we need to make some additional transformations on the bounding boxes, confidence and class probabilities to obtain either the loss value or the final predictions.

To decode the bounding boxes, YOLO[15] uses the following transformation:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \tag{2.14}$$

where:

- t_x, t_y, t_w, t_h: Predicted bounding box coordinates
- c_x, c_y: Offset from the top left corner of the image (fractional form)
- p_w, p_h: Bounding box priors

- σ: Sigmoid function

Then, for the confidence and class probabilities we simply apply the sigmoid function:

$$\begin{aligned} Obj_{Conf} &= \sigma(t_o) \\ Pr(Class) &= \sigma(t_c) \end{aligned} \tag{2.15}$$

Finally, at the inferring time, the bounding boxes are multiplied by the network stride and the predictions are stacked into a single tensor (the model's output).

2.2.3 Loss

YOLO uses a multi-part loss function. It might look a bit daunting at first but in reality, it is just an aggregation of multiple losses.

$$L = L_x + L_y + L_w + L_h + L_{conf} + L_{cls} \tag{2.16}$$

where:

- L_x, L_y: Loss for the x and y bounding box offsets
- L_w, L_h: Loss for the *width* and *height* bounding box offsets
- L_{conf}: Loss for the confidence of each bounding box
- L_{cls}: Loss for the classifier

The bounding box is regression problem so we use a regression loss. Specifically, the authors used the Mean Squared Error (MSE) for all the box components (x, y, w, h):

$$\begin{aligned} L_x &= MSE \\ L_y &= MSE \\ L_w &= MSE \\ L_h &= MSE \end{aligned} \tag{2.17}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x_i})^2 \tag{2.18}$$

Then, we need to compute the loss for the object confidence and class probabilities. Both are classification problems, so a simple cross-entropy loss will do the work. Since we are dealing just with two classes we can use a modified version of the Cross-Entropy (CE) loss, the Binary Cross-Entropy (BCE).

$$BCE = -(y\log(p) + (1-y)\log(1-p)) \tag{2.19}$$

Back to the confidence loss, it is made up of two components: i) the object confidence and ii) the no-object confidence

$$L_{conf} = \alpha \cdot L_{conf\_obj} + \beta \cdot L_{conf\_no\_obj} \tag{2.20}$$

where:

α and β are two parameters to control the importance of each of the sub-losses (a sort of hard negative mining).

Now, these sub-losses are computed using the Binary Cross-Entropy of the confidence for both, objects and no-objects.

$$\begin{aligned} L_{conf\_obj} &= BCE \\ L_{conf\_no\_obj} &= BCE \end{aligned} \tag{2.21}$$

Similarly, we compute the classification loss taking the class predictions of the objects (there is no need for no-objects as we don't care for the background).

$$L_{cls} = BCE \tag{2.22}$$

Finally, by doing some arrangements and minor modifications for numerical stability, we can combine all the above losses into a final objective function:

$$\begin{aligned} L &= L_{loc} + L_{conf} + L_{cls} \\ &= \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\ &+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\ &+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\ &+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \end{aligned} \tag{2.23}$$

where:

- $\mathbb{1}^{obj}$: denotes if object appears in cell i
- $\mathbb{1}_{ij}^{obj}$: denotes that the j-th bounding box predictor in cell i is "responsible" for that prediction

2.2.4 Anchor boxes

Anchor boxes improved the recall in YOLO by a 7% margin[13], but instead of hand-pick the priors as other researchers did[11][4], they clustered the ground-truths by running K-means over the dataset. As a result, they could find good priors automatically (see Figure 2.7).

It is important to point out that the anchors are not arbitrary, but the number of clusters up to some extent is because this number is a trade-off for the recall and the complexity of the model.

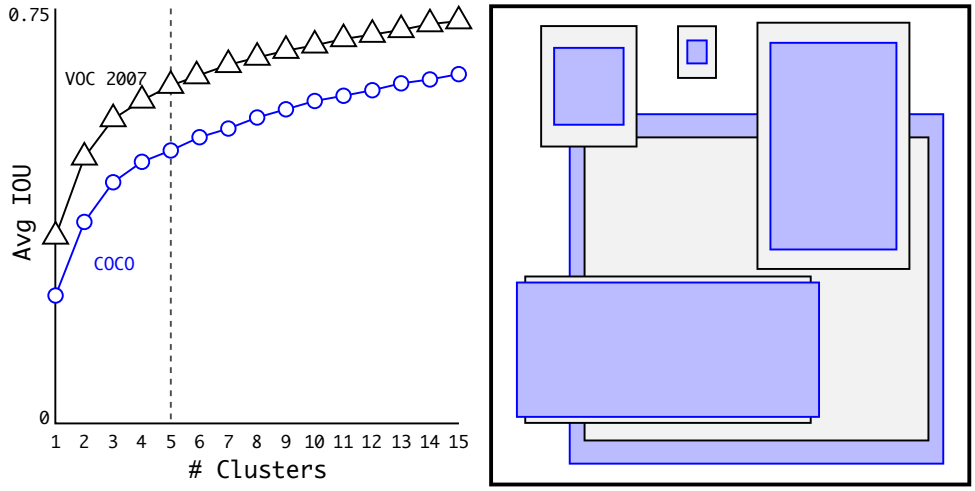


Figure 2.7: **Clustering box dimensions**: We use K-means to clusters the bounding boxes to get good priors automatically [13]

One problem that arises here is that K-means is typically used with the Euclidean distance as it works perfectly fine for many cases. But for this particular task, using the Euclidean distance meant that larger boxes would generate more error than the smaller ones. To fix this, the authors derived a metric distance based on the IoU to get priors with good IOU scores:

$$d(box, centroid) = 1 - IoU(box, centroid) \tag{2.24}$$

Finally, the authors use $k = 5$ as it seemed a good trade-off for the recall and the complexity of the model.

Chapter 3

Methodology

3.1 Definitions

3.1.1 Bounding box

A bounding box is a box that wraps an object around. There are many representations but the two most popular are:

- **Boundary coordinates:** $(x_{min}, y_{min}, x_{max}, y_{max})$
- **Center-size coordinates:** (c_x, c_y, w, h)

For the sake of simplicity, it is recommended to use a fractional form so that the regions are independent of the image size.

3.1.2 Jaccard Index

The Jaccard Index, also known as Intersection Over Union (IOU) or Jaccard Overlap, measures the degree or extent to which two boxes overlap.

It can be computed as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.1}$$

An IoU of 1 means that the two boxes are indeed the same box, while an index of 0 indicates they're two completely different boxes with no overlapping at all (mutually exclusive spaces).

3.1.3 Multibox

Multibox is a technique derived from[8] for detecting objects where the prediction consists of two parts:

- **Regression**: Box coordinates that may or may not contain an object.
- **Classification**: Class scores for the object types (including the background).

3.1.4 Confusion matrix

The confusion matrix, also known as an error matrix, is a table to visualize the performance of a classifier. It has two rows and two columns that contain the number of false positives, false negatives, true positives, and true negatives.

		Predicted	
		Positive	**Negative**
Actual	**Positive**	TP	FP
	Negative	FN	TN

Table 3.1: **Confusion matrix for binary classification**

For the specific case of object detection, getting those values might be a bit tricky since depends on the framework we use to define which prediction is correct, and which doesn't. Nevertheless, we can agree with their theoretical meaning.

- **True Positive (TP)**: A correct detection.
- **False Positive (FP)**: A wrong detection
- **False Negative (FN)**: A ground truth not detected
- **True Negative (TN)**: When the background is detected as a background[1]

In practice, we mostly care for the correct detections (TP) and the incorrect ones (FN) since most object detectors don't detect the background (at least explicitly).

3.1.5 Precision

The precision is the fraction of relevant detections that were detected over all the detections.

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{All \quad detections} \tag{3.2}$$

3.1.6 Recall

The recall is the fraction of relevant detections that were detected over all the ground truth detections.

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{All \quad ground - truths} \tag{3.3}$$

3.1.7 F1-Score

The F1-score is the harmonic mean of Precision and Recall. A metric that relates both ratios into a single score.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3.4}$$

[1] It does not usually apply

3.2 Singularities of mathematical expression detection

From a technical perspective, both, general-object detection (trees, cars, humans,...) and mathematical-expression detection are the same thing. However, the latter presents some singularities that make its detection harder.

For instance, in general-object detection, the objects to detect are usually quite large when compared to the size of the input image (see Figure 3.1), even small objects are relatively big. This results in coarse-grained detectors that work really well for this type of objects but failed when detailed is needed or the objects look quite similar (e.g. car models, dog breeds,...).

Another point that eases the work of a general-object detector is the color. Some objects are quite similar in shape but completely different in terms of colors (i.e. limes and lemons). Therefore, if we allow the model to take into account the color, many of these objects will be more easily discriminated.

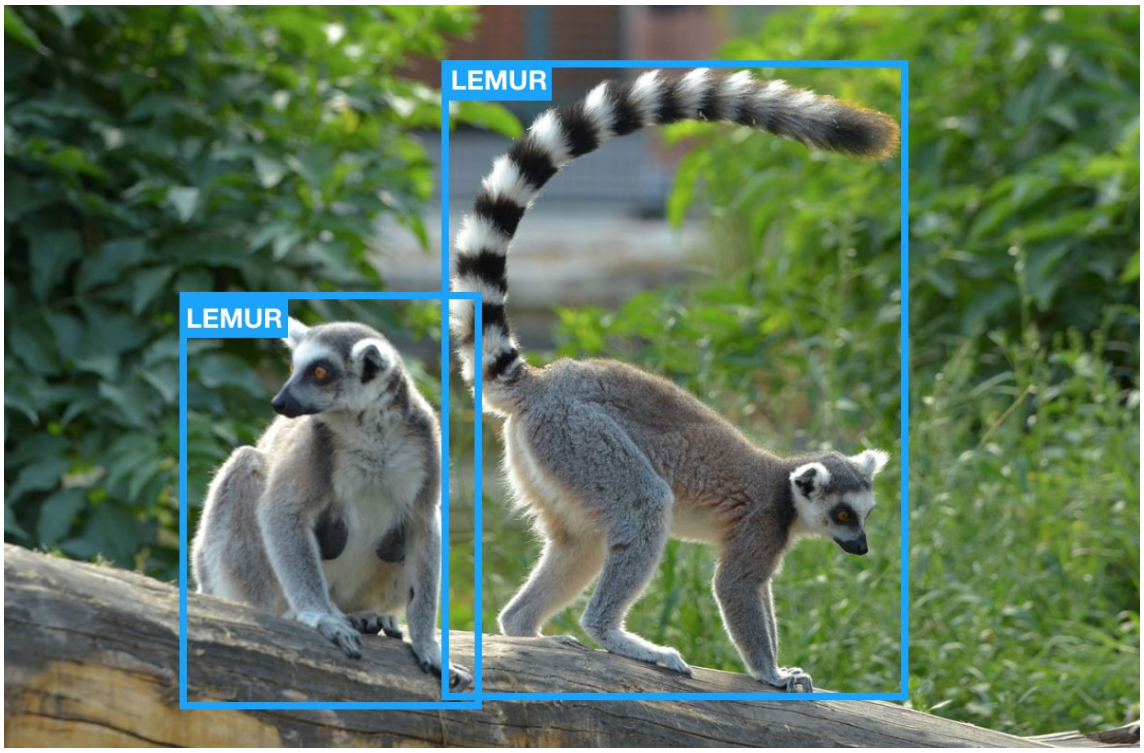
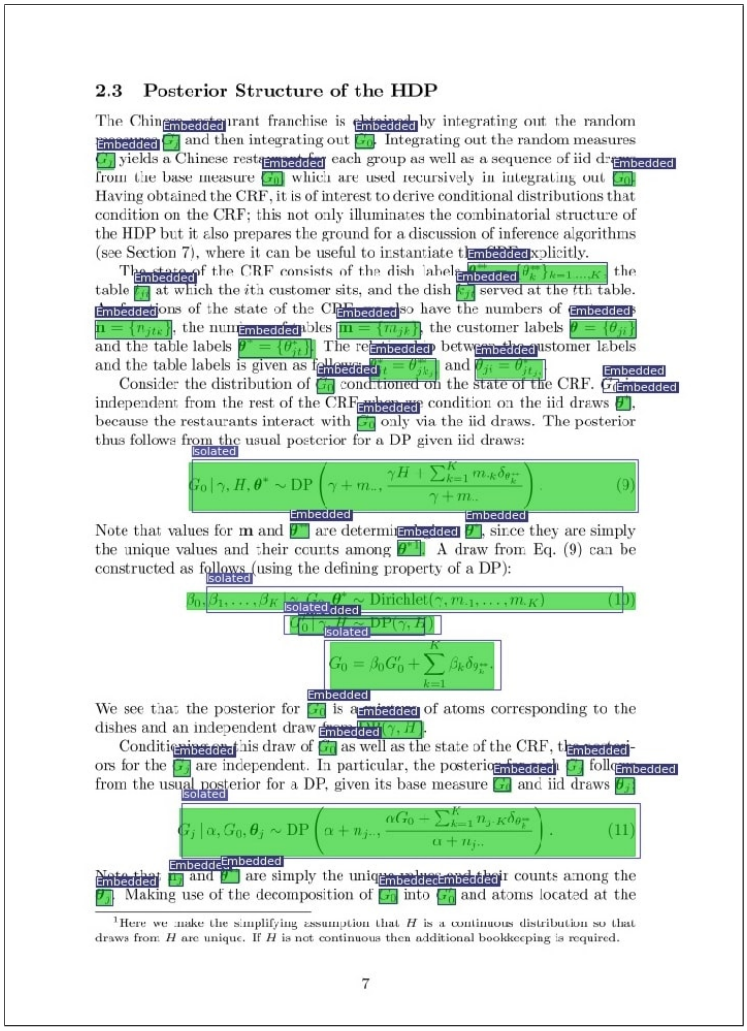


Figure 3.1: **General-object detection problem**

On the other hand, for the task of mathematical-expression detection, we need to detect big and really small objects such as isolated equations or character-level expressions (see Figure 3.2). Besides the size, these objects might look quite similar even for humans. For instance, embedded expressions can be easily mistaken as text, especially when they are one-character long (e.g x, x and x).

This nuance has a strong impact on the performance of a detector because, to begin with, we need a high-resolution classifier (remember that when detecting big objects we don't need as much resolution as when detecting small objects). Then, we need way more hypothesis as objects can be one-character long. And finally, we need better and more accurate feature extractors to detect the nuances previously commented. Furthermore, most of the state-of-the-art detectors are better at detecting big objects than small ones

due to this efficiency constraint so we need to modify the state-of-the-art architecture in order to make them suitable for our needs.



2.3 Posterior Structure of the HDP

The Chinese restaurant franchise is obtained by integrating out the random measure G_j and then integrating out G_0. Integrating out the random measures G_j yields a Chinese restaurant for each group as well as a sequence of iid draws from the base measure G_0 which are used recursively in integrating out G_0. Having obtained the CRF, it is of interest to derive conditional distributions that condition on the CRF; this not only illuminates the combinatorial structure of the HDP but it also prepares the ground for a discussion of inference algorithms (see Section 7), where it can be useful to instantiate the variables explicitly.

The state of the CRF consists of the dish labels $(\theta^{**}_k)_{k=1,\ldots,K}$, the table t_{ji} at which the ith customer sits, and the dish k_{jt} served at the tth table. Implicit in the state of the CRF are also have the numbers of customers $\mathbf{n} = \{n_{jtk}\}$, the number of tables $\mathbf{m} = \{m_{jt}\}$, the customer labels $\boldsymbol{\theta} = \{\theta_{ji}\}$ and the table labels $\boldsymbol{\theta}' = \{\theta'_{jt}\}$. The relationship between the customer labels and the table labels is given as follows: $\theta_{ji} = \theta'_{jt_{ji}}$ and $\theta'_{jt} = \theta^{**}_{k_{jt}}$.

Consider the distribution of G_0 conditioned on the state of the CRF. G_0 is independent from the rest of the CRF and we condition on the iid draws $\boldsymbol{\theta}^*$, because the restaurants interact with G_0 only via the iid draws. The posterior thus follows from the usual posterior for a DP given iid draws:

$$G_0 \mid \gamma, H, \boldsymbol{\theta}^* \sim \mathrm{DP}\left(\gamma + m_{\cdot\cdot}, \frac{\gamma H + \sum_{k=1}^{K} m_k \delta_{\theta^{**}_k}}{\gamma + m_{\cdot\cdot}}\right). \quad (9)$$

Note that values for $\mathbf{m}$ and $\boldsymbol{\theta}^*$ are determined by $\boldsymbol{\theta}$, since they are simply the unique values and their counts among $\boldsymbol{\theta}$.[1] A draw from Eq. (9) can be constructed as follows (using the defining property of a DP):

$$(\beta_0, \beta_1, \ldots, \beta_K) \mid \gamma, m_{\cdot 1}, \ldots, m_{\cdot K} \sim \mathrm{Dirichlet}(\gamma, m_{\cdot 1}, \ldots, m_{\cdot K}) \quad (10)$$

$$G'_0 \mid \gamma, H \sim \mathrm{DP}(\gamma, H)$$

$$G_0 = \beta_0 G'_0 + \sum_{k=1}^{K} \beta_k \delta_{\theta^{**}_k}$$

We see that the posterior for G_0 is a mixture of atoms corresponding to the dishes and an independent draw $G'_0 \sim \mathrm{DP}(\gamma, H)$.

Conditioning on this draw of G_0 as well as the state of the CRF, the posteriors for the G_j are independent. In particular, the posterior for G_j follows from the usual posterior for a DP, given its base measure G_0 and iid draws $\boldsymbol{\theta}_j$:

$$G_j \mid \alpha, G_0, \boldsymbol{\theta}_j \sim \mathrm{DP}\left(\alpha + n_{j\cdot\cdot}, \frac{\alpha G_0 + \sum_{k=1}^{K} n_{j\cdot k} \delta_{\theta^{**}_k}}{\alpha + n_{j\cdot\cdot}}\right). \quad (11)$$

Note that $\mathbf{m}$ and $\boldsymbol{\theta}^*$ are simply the unique values and their counts among the $\boldsymbol{\theta}_j$. Making use of the decomposition of G_0 into G'_0 and atoms located at the

[1] Here we make the simplifying assumption that H is a continuous distribution so that draws from H are unique. If H is not continuous then additional bookkeeping is required.

7

Figure 3.2: **Mathematical-expression detection problem**

It is worth to mention that this is a complex problem and the model trained in using documents will have lots of problems if we try to detect handwritten equations.

3.3 Topology adjustments

SSD[4] and YOLO[15] are state-of-the-art object detectors for general-object detection, or more precisely, state-of-the-art detectors for COCO[25], Pascal VOC[2] and ImageNet-like[26] datasets.

Following the principles of transfer learning, a model that worked well for a generic task should also work relatively well for a different but closely related task. However, this does not mean that those models should work in a plug-and-play manner for a new domain. What usually happens (and what we had to do) is that we need to modify their architecture so that they can be trained on the new problem in an efficient and performant way.

The specific modifications of both models will be discussed in a later section but as a brief introduction, we will say that SSD[4] suffers from memory-related problems when

the confidence in a detection is low (the IoU of many hypotheses may lead to a memory overflow), numerical instabilities during training, intrinsic difficulties when detecting small objects and very sensitive parameterization. On the other hand, YOLOv3[15] behaves pretty without performing any modification since it has fewer problems with small objects than big ones. Most modifications in YOLO are focused on improving its baseline accuracy for a given task (i.e. feature extractor for fine-grained detail, specific anchors boxes for your problem, finding a good parametrization,...).

3.4 Data processing

3.4.1 Pre-processing

As the saying goes: *"Data is the new oil"*. Data allows us to better understand the world in which we live, providing us with knowledge, making predictions and automating tasks. Until recently, many tasks were reserved only for humans, but with the help of machine learning models, we can now solve complex problems that a few days ago seemed completely impossible to automate.

The main problem of these models is that they usually need lots of data, and data is not only a scarce resource but an expensive one.

To mitigate this problem, we use data augmentation. A technique that allows us to increase the size of our limited dataset to infinity, through the use of small transformations. Thus, machine learning models extract every ounce of knowledge from our original dataset.

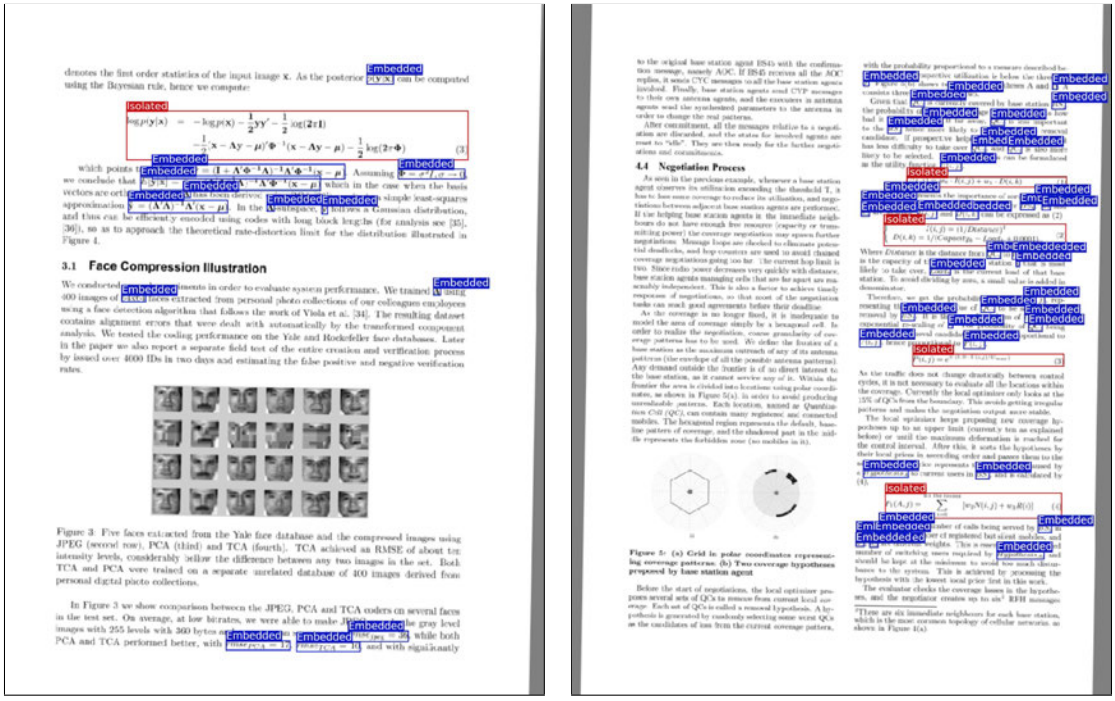


Figure 3.3: **Data augmentation**

In practice, this data augmentation cannot be performed indefinitely since models need new samples from which to learn new and relevant features. For the specific case of mathematical expression detection, we have performed a standard data augmentation (rotation, scaling, shifts, padding,...) as can be seen in Figure 3.3.

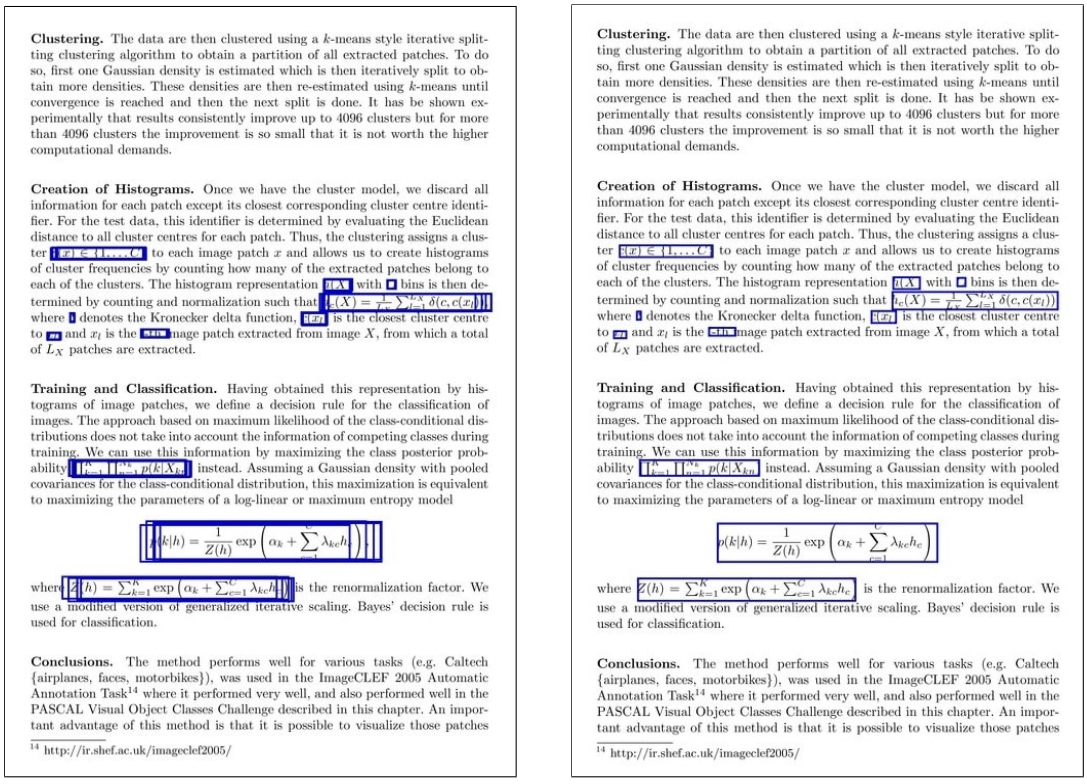
As we need high-resolution images, in order to normalize the size of the images we decided to use the minimum height that allowed us to clearly see the individual characters in a document. As images can have different aspect ratios, we set the width as the maximum width found in our dataset, once all images were resized to the height established. Then, for those images that did not reach the normalized width, we applied a padding and centered the image horizontally. To bring some numbers up, the reference height and width were 1024px and 800px, respectively. Although we tried with higher resolutions, we always preserved that aspect ratio and data augmentation.

3.4.2 Post-processing

Non-Maximum Suppression

Object detectors rarely predict one single bounding box per object. To filter the extra overlapping boxes we need to apply a post-processing step called Non-Maximum Suppression (see Figure 3.4).

Non-Maximum Suppression (NMS) is a post-processing technique to transform a smooth response map that triggers many imprecise object window hypotheses into, a single bounding-box for each detected object[27].



Clustering. The data are then clustered using a k-means style iterative splitting clustering algorithm to obtain a partition of all extracted patches. To do so, first one Gaussian density is estimated which is then iteratively split to obtain more densities. These densities are then re-estimated using k-means until convergence is reached and then the next split is done. It has be shown experimentally that results consistently improve up to 4096 clusters but for more than 4096 clusters the improvement is so small that it is not worth the higher computational demands.

Creation of Histograms. Once we have the cluster model, we discard all information for each patch except its closest corresponding cluster centre identifier. For the test data, this identifier is determined by evaluating the Euclidean distance to all cluster centres for each patch. Thus, the clustering assigns a cluster $c(x) \in \{1, \ldots, C\}$ to each image patch x and allows us to create histograms of cluster frequencies by counting how many of the extracted patches belong to each of the clusters. The histogram representation $t(X)$ with $\square$ bins is then determined by counting and normalization such that $t_c(X) = \frac{1}{L_X} \sum_{l=1}^{L_X} \delta(c, c(x_l))$ where δ denotes the Kronecker delta function, $t(x_l)$ is the closest cluster centre to x_l and x_l is the $\square$ image patch extracted from image X, from which a total of L_X patches are extracted.

Training and Classification. Having obtained this representation by histograms of image patches, we define a decision rule for the classification of images. The approach based on maximum likelihood of the class-conditional distributions does not take into account the information of competing classes during training. We can use this information by maximizing the class posterior probability $\prod_{k=1}^{K} \prod_{n=1}^{N_k} p(k|X_{kn})$ instead. Assuming a Gaussian density with pooled covariances for the class-conditional distribution, this maximization is equivalent to maximizing the parameters of a log-linear or maximum entropy model

$$p(k|h) = \frac{1}{Z(h)} \exp\left(\alpha_k + \sum_{c=1}^{C} \lambda_{kc} h_c\right)$$

where $Z(h) = \sum_{k=1}^{K} \exp\left(\alpha_k + \sum_{c=1}^{C} \lambda_{kc} h_c\right)$ is the renormalization factor. We use a modified version of generalized iterative scaling. Bayes' decision rule is used for classification.

Conclusions. The method performs well for various tasks (e.g. Caltech {airplanes, faces, motorbikes}), was used in the ImageCLEF 2005 Automatic Annotation Task[14] where it performed very well, and also performed well in the PASCAL Visual Object Classes Challenge described in this chapter. An important advantage of this method is that it is possible to visualize those patches

[14] http://ir.shef.ac.uk/imageclef2005/

(a) Before NSM (b) After NSM

Figure 3.4: **Non-Maximum Suppression**

There are many NMS methods that work for special cases like rotated boxes o polygons, but as our case was a simple one, we decided to implement our own version.

Basically, it works as follows:

1. Remove all boxes with a low confidence score

2. Combine all the overlapping boxes into a new one, weighting their coordinates by their confidence.

Additionally, we have enabled a flag to only keep the box with the highest confidence score.[2]

3.5 Evaluation

3.5.1 Prediction correctness

To evaluate an object detector we need to know if the predicted detection was correct or not. The problem is that this is a tricky task because object detection is more open-ended than other fields.

For instance, if a predicted box doesn't intersect at all with any ground-truth we can confidently say that the detection is wrong. But what happens if it intersects just a 10%, or a 30%? Could we confidently says that the detection is correct? Or even worse, what happens if a blind detector adds a box wrapping the whole input image? All the ground-box intersections will have a 100% but the detection couldn't be more wrong.

To address these issues, we have introduced at the beginning of the chapter some definitions that might help such as the Jaccard index, the precision, recall, f1-score, etc. However, there are still some loose ends to know if a prediction is correct or not.

Unlike the typical regression or classification task where each input-output is matched with a target in a relationship 1-to-1, in object detection, there is no clear matching between the input-output and the targets. It is clear that for each image there are n ground-truths, but the output of a model is a list of predictions with no target assigned (see Figure 3.5).

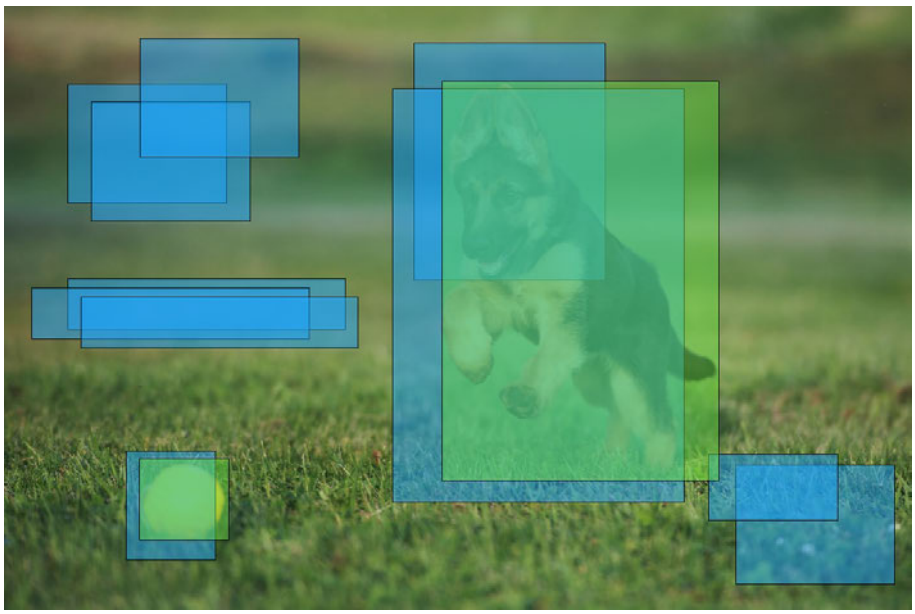


Figure 3.5: **Prediction matching**: In blue the predictions, and in green the ground-truths

[2]Since this is a simple algorithm, we have considered that the exact description of the method is not needed.

To match each prediction with a ground-truth, we compute the overlapping coefficient of each prediction with all the ground-truths using the Jaccard Index. Then, each prediction is assigned to the ground-truth that has the best overlap (a ground-truth can have many predictions assigned). If they both belong to the same class and the best overlapping in greater than a certain threshold (usually 50%), we label that prediction as *correct*.

$$correctness(b_i^p) = \begin{cases} 1, & if \quad c_i^p = c_j^g \quad and \quad IoU(b_i^p, b_j^g) > 0.5 \\ 0, & otherwise \end{cases} \tag{3.5}$$

where:

- b_i^p: Predicted box i
- b_j^g: Ground-truth box j
- c_i: class assigned to box i

However, to label each *non-correct* prediction as *incorrect* is a bit harsh since there are many subtleties. That is why Hoiem et al.[16] designed a better methodology to diagnose the error in object detectors. Here, we used the following simplified version:

- **Correct**: Correct class and $IoU > 0.5$
- **Localization**: Correct class and $1.0 <= IoU > 0.5$
- **Other**: Incorrect class and $IoU > 0.1$
- **Background**: $IoU <= 0.1$

3.5.2 Metrics

As most of the object detection competitions use the Mean Average Precision (mAP) or a derived one for their main metric, this will be the reference metric in our study. In addition to this, it's a very popular one amongst researchers but it has some problems that we will discuss later.

Precision-Recall Curve

The recall and precision ratios are closely related to the confidence threshold. For instance, when the confidence in a prediction decreases, the recall tends to monotonically increase while the precision goes down. This is what usually happens, but there might many reasons why this does not occur.

What is important here, it's the relationship between the confidence threshold and the recall-precision pairs. By setting the confidence threshold at different levels, we get different pairs of values and if we plot these values, we obtain the *Precision x Recall* curve.

Thanks to this curve we can now evaluate the performance of an object detector.

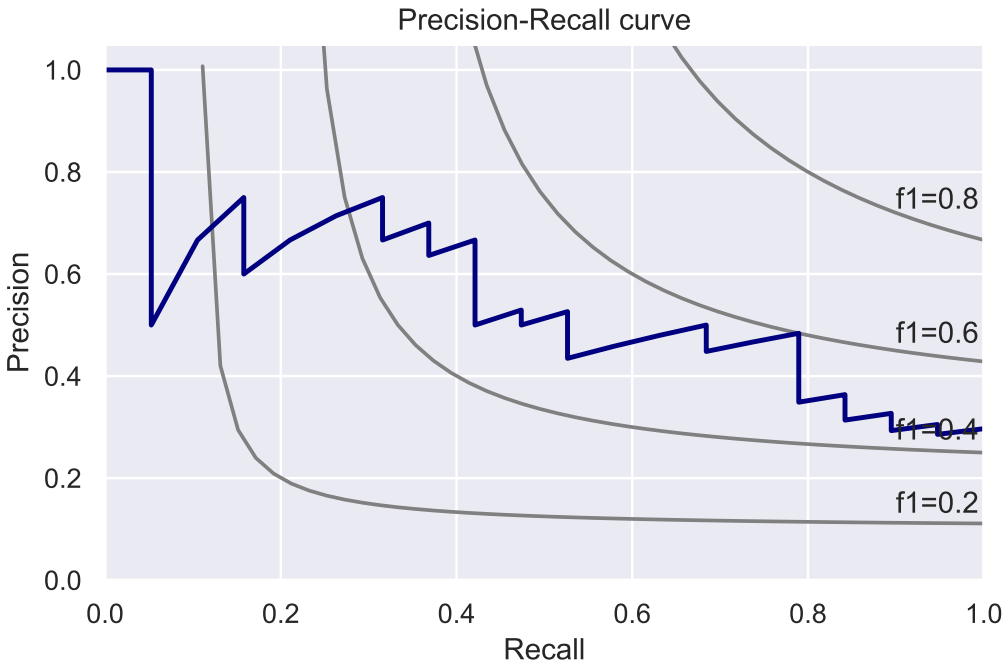


Figure 3.6: **Precision-Recall curve**: In blue the precision-recall curve, and in gray the ISO-f1 lines.

Average Precision

The Precision-Recall curve gives us a lot of information about the performance of an object detector, but there are a few problems with it. First, this curve zig-zags a lot and that makes it hard to compare the performance of different detectors. Secondly, it's a curve, not a score. We want a single score to know how good is our detector.

To transform this zig-zag plot into a single value we need to compute the Area Under the Curve (AUC). Since the pair of values we have are discrete, we don't have a "curve" in the strict sense so we need to interpolate its values.

The two most popular interpolations are:

- **11-point interpolation**: Summarizes the shape of the curve by averaging the precision at a set of 11 equally spaced recall levels.[3]
- **Interpolating all points**: Instead of interpolating over 11 equally spaced points, it interpolates through all points.[4]

To compute the AUC we approximate the previous curve using steps. This means that for each recall value, we interpolate precision by taking the maximum precision whose recall value is greater than its current recall value. Then, to compute the area we only have to sum the areas of the multiple *boxes* we have obtained from the stepped-curve. This can be easily understood looking at Figure 3.7.

[3]This one was used by PASCAL VOC until 2010.
[4]Full interpolation is the metric currently used in the PASCAL VOC Challenge

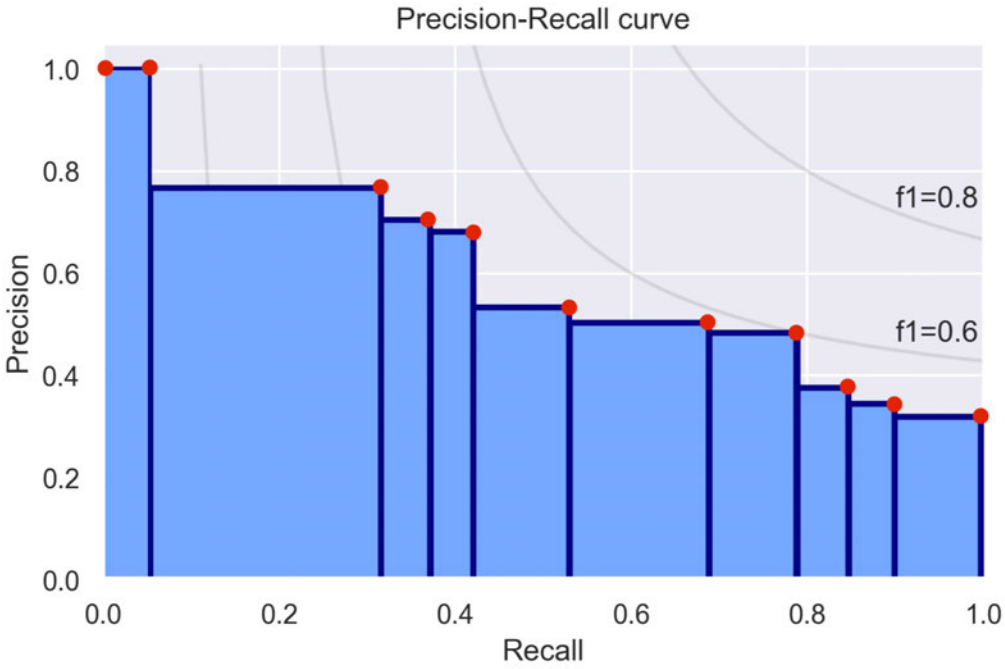


Figure 3.7: **Interpolating all points**: All points are interpolated to compute the precision-recall area under curve (PR AUC)

Finally, it is worth to point out that in this specific context, the *Average Precision* is the AUC itself.

Mean Average Precision

The Average Precision works directly as an object detector metric if there is just one class (and the background). But what happens when we have multiple classes? Easy! We compute the mean of the *Average Precision* for each class.

$$mAP = \frac{1}{N} \sum_{i}^{N} AP_{class_i} \tag{3.6}$$

3.5.3 ROC curve

The Receiver Operating Characteristic curve, or ROC curve, shows the ability of a binary classifier to discriminate among classes when its discrimination threshold is varied.

It is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings (see Figure 3.8).

This curve needs the true binary labels and the probability (or confidence) scores for each prediction. To get these scores is rarely a problem, but getting the true binary labels is a bit tricky as we have discussed earlier. Anyway, by setting a good correctness framework, this could shouldn't be hard to get.

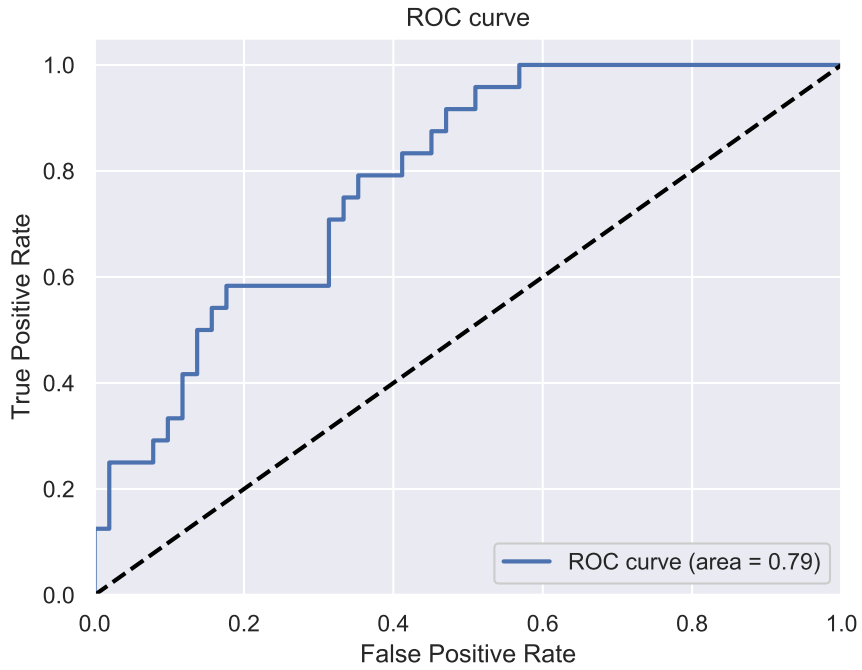


Figure 3.8: **ROC curve**: In blue the curve of a binary classifier, and in black baseline "curve" of a random classifier.

3.5.4 Discussion

An important problem in the field of object detection is the lack of consensus to choose an evaluation metric that measures the performance of the detectors.

As it is been said before, many researchers use the Mean Average Precision as well as the most popular object detection competitions[5].

Generally speaking, one of the most important things in research is to be able to compare your results with other people. To do a fair comparison we all need to play under the same set of rules. One way to do this is to use the same metrics as other authors did. The problem of this approach is that it is not always clear how a metric was exactly implemented or which evaluation methodology was used. Maybe there was a minor error in the implementation that biased the results, or if the results come from a competition and this is not open anymore, or the metrics used are unknown, etc. In other words, there are many problems when comparing algorithms but from my point of view, popular competitions are great because the evaluation depends on reliable third-parties and besides you can compare your models with others that are playing under the exact same rules.

On the other hand, there is an additional problem with most metrics because they might not be optimal in many cases. That is because they usually emphasize some aspects in exchange for de-emphasizing others. For instance, the COCO metric favors better bounding boxes at the expense of classification accuracy. And as Joseph Redmon et al. stated in YOLOv3[15]: *"Is there a good reason to think that more precise bounding boxes are more important than better classification? A miss-classified example is much more obvious than a bounding box that is slightly shifted."*

[5]COCO[25] and PASCAL VOC[2]

For instance, the problem of the Mean Average Precision is that the only thing that matters is the per-class rank-ordering so two hypothetical detectors might get similar results whereas one of them is clearly giving bad results. See in Figure 3.9 the predictions from two hypothetical detectors where one bounding box is completely miss-classified.

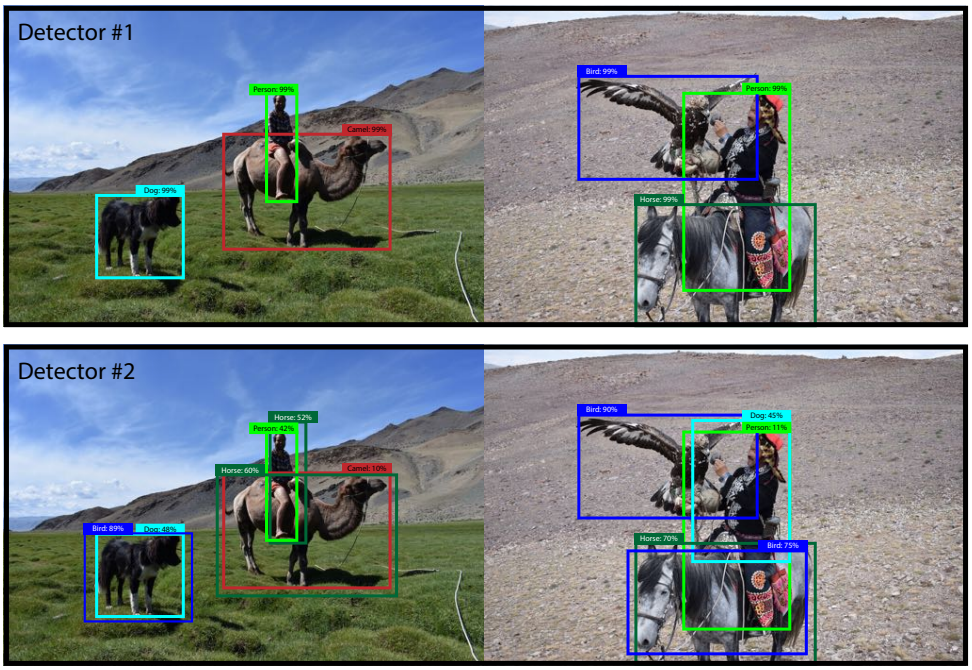


Figure 3.9: **Two identical detectors according to mAP** [15]

Similarly, there are also problems with the IoU metric. Russakovsky et al.[28] reported that that humans have a hard time distinguishing an IOU of 0.3 from 0.5. Thus, which evaluation metric should we use is still an open question that might depend on specific the case.

Nowadays, the trend is to use masks instead of bounding boxes whenever is possible due to the high level of detail that you get. Nevertheless, for the specific problem of mathematical expression detection, it seems like bounding boxes are perfectly suited for it as is the most natural form to establish bounding regions.

Chapter 4

Experiments and results

4.1 Dataset description

For this study, we have used the *Marmot dataset for math formula recognition*[29], a ground-truth dataset for mathematical formula identification.

This dataset is a collection of PDF documents downloaded from *CiteSeerX*. It contains all the original PDF files, their corresponding document images, the metadata associated with these documents and the ground-truths for the mathematical formulas along with the ground-truths for the objects embedded in them.

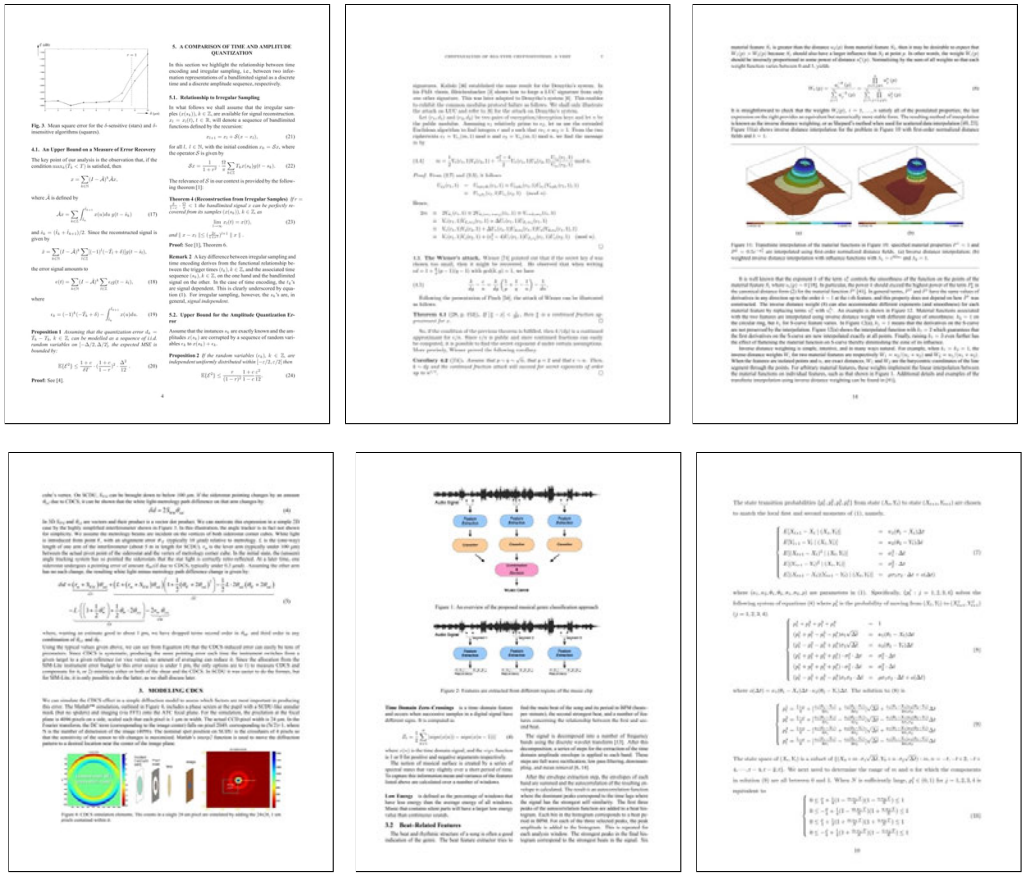


Figure 4.1: **Math Formula Recognition dataset**: Six random samples from the Marmot dataset[29]

Some interesting numbers on the dataset are:

- Dataset version: 1.0
- Specifics

  Content: 400 scanned pages in JPEG format (from 194 PDF documents)

  Dimensions: ~4250x5500

  Color space: RGB and Gray

  Isolated formulas: 1575

  Embedded formulas: 7907

  Ground-truth format: XML

- Size: 736MB compressed; 1.17GB uncompressed
- Compression format: *ZIP*
- Link: `http://www.icst.pku.edu.cn/cpdp/docs/20190424192347869700.zip`

4.2 Experiments

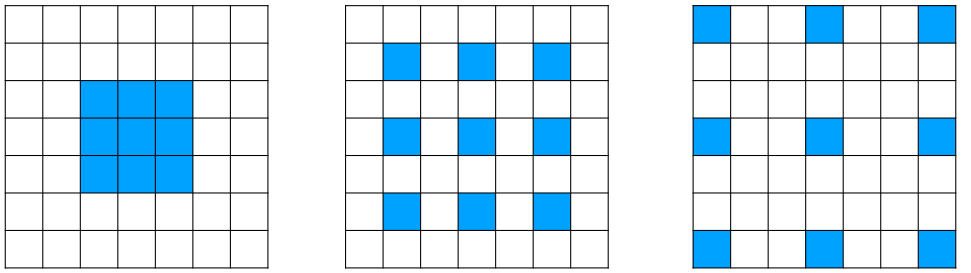
4.2.1 SSD

Network design

The network presented in the paper[4] was designed for the PASCAL VOC, COCO, and ILSVRC datasets; fixing the input size to 300x300 and 512x512. For these datasets, those resolutions worked well, but as we are dealing with characters in scanned pages, we have to increase the input size. This increase in the image resolution has a strong impact on the network performance as well as on the memory consumed.

To overcome these issues we introduced several modifications to the original network architecture but keeping VGG16[17] as the base feature extractor. First, we used high-resolution images to capture the character-level detail needed for the task. This meant to increase the input size from 300x300 to 1024x1024 (pixels) but at the expense of generating a huge amount of priors (around 100,000). As we cannot reduce the input resolution we decided to reduce and crop part of the white margins that contain no information. Finally, the input size used was 1024x800.

Although the number of priors is now smaller, it is still too high to be considered efficient. To further decrease the number of predictions, we subsample every feature map along a particular dimension using the kernel dilation trick in a process known as decimation.



(a) Kernel 3x3; Dilation 0 (b) Kernel 3x3; Dilation 1 (c) Kernel 3x3; Dilation 2

Figure 4.2: **Dilated convolutions**: Blue points represent active locations of a 3x3 convolutional kernel

In addition to this, we have also brought down the number of priors per feature map location from 4-6 to 1-6 in a pyramidal style. Smaller feature maps have fewer priors per location than the big ones (this will be explained in detail in the next section).

Then, once we have the final output containing all the predicted regions, we filter out those that fall under a certain threshold as other methods do. But now, we have introduced a hard limit in which we sort all predictions by their confidence, keeping just the top N with the highest scores.[1] This is previous to the post-processing steps so that the next steps receive an input tensor of manageable size.

Building priors

The authors of SSD[4] used a grid of carefully studied parameters to construct their set of priors. Given the high-resolution needed for this task, the default number of priors was too high and we couldn't train the model following that approach due to memory limitations.

In the Marmot dataset[29], many of the embedded equations are just one or two-character long and many of the isolated equations take all width of the document. Taking this observation into account we designed similar a method to the one presented in the paper[4] but based on clustering the aspect ratios w.t.r the average IoU using K-Means as YOLOv2[13] does.

After this clustering, we approximate the aspect ratios to integer numbers. Now we had less but better priors. Nevertheless, there were still too many to train the model. To overcome this issue, we assigned the clustered aspect ratios taking into account the scale in which they tend to occur. For instance, the 1:1 embedded equations are pretty common on the small scales, but quite unusual in the big ones. This allowed us to optimize the aspect ratio for each scale and greatly reduce the number of priors used since most priors come from feature maps that account for small objects.

This is because big feature maps have more unit cells but with a smaller receptive field. A feature map of 128x128 has 16,384 sensing units, but one with half the size (64x64) has four times fewer units (4,096).

Training

Training SSD[4] for mathematical expressions using this very customized high-resolution detector has been difficult. Partly because SSD[4] is very sensitive to the bounding box

[1] This is different from the usual *top_k* parameter which returns the top k predictions of the final result in order to not overpopulate the final view.

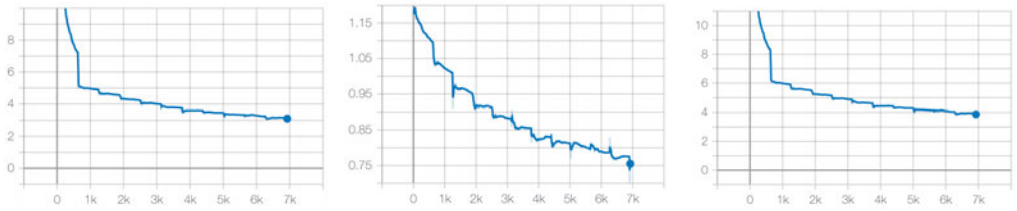
size, showing a much worse performance on smaller objects than bigger ones[4]. The other reason is because of the compromises that we had to do to bring down the number of predictions into a manageable size which introduced additional problems.

As it has been said in previous sections, we had to use high-resolution images for our detector but leaving the default architecture meant too many priors. So to further bring down the number of priors we had to half some feature maps and reduce the number of anchors.

These compromises are not a model's problem but a memory-related one. When we check the overlapping between detections and priors boxes, we build a matrix $N * M$ of IoUs. If these numbers are too big, the resulting matrix will occupy a big chunk of our memory, which added to the memory required by the model itself and other GPU tensor operations, will result in a not enough memory error.

Initially, we train the model from scratch but as the results we relatively poor in terms of recall, we decided to try with transfer learning to keep tuning its hyper-parameters.

First, we initialized our feature extractor using the pre-trained weights from VGG16[17] available in PyTorch[30] and Xavier Initialization[31] for the remaining layers. When our network was initialized, then we train it using the dataset from the Pascal VOC 2007[2] competition plus standard data augmentation techniques such as flips, rotations, scaling,...



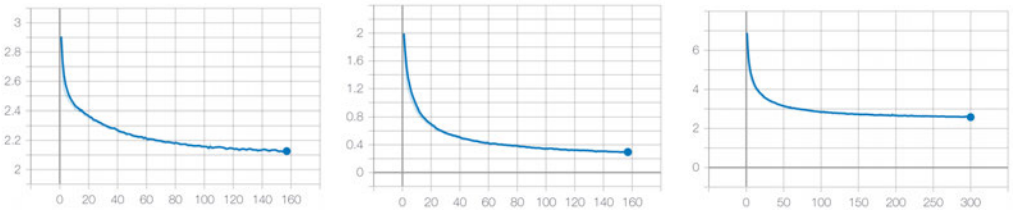
(a) Confidence loss (b) Localization loss (c) Total loss

Figure 4.3: **Multi-part loss of SSD from the Pascal VOC dataset**

Looking at Figure 4.3 we see that all losses decrease (it's multi-part loss) although they are not as smooth as they should. Since we are not training for VOC but looking for a good initialization for our model, we decided not to train for many iterations. Once the model was trained, we double-checked the results by visualizing the predicted objects (see Figure 4.4).

Figure 4.4: **SSD pre-trained on the VOC Pascal dataset**

Then, we re-train the model for the Marmot dataset[29] but using these weights from VOC[2]. As we have seen from Figure 4.5 the model converges pretty smoothly but the results are still bad in terms of recall.



(a) Confidence loss (b) Localization loss (c) Total loss

Figure 4.5: **Multi-part loss of SSD from the Marmot dataset**

The particularities of each training depend on the specific experiment but in general, we have followed the same approach all models. Full images as the input preserving the aspect ratio (adding padding if necessary), Adam[32] as the stochastic function optimizer with a learning rate of 1e-4, a maximum of 300 epochs or 20h of training, 90% of the training data for training and 10% for validation, small batch sizes ranging from 1 to 2 due to memory limitations in our GPU, 1 to 2 gradient accumulations per optimization step, hard negative mining and a relatively standard data augmentation.

For the data augmentation, we used the following transforms:[2]

- **Shift:** $[-0.0625, +0.0625]$
- **Scale:** $[-0.0, +0.0625]$
- **Rotate:** $[-2, +2]$
- **Interpolation:** Bilinear interpolation with coefficients (1, 0)[3]
- **Border mode:** Replicate
- **Color:** RGB (3-channel) or Gray (1-channel)
- **Resize mode:** Keep aspect ratio
- **Longest max. size:** 1024

Results

The results showed in Table 4.3 are computed using an IoU of 0.5, a confidence threshold of 0.5, a non-maximum suppression of 0.3 and the mean average precision has been computed over 11 points.

These results will be discussed in the next section.

[2]Some of them might vary depending on the experiment.
[3]*INTER_AREA* from OpenCV

Trial	Precision	Recall	F1	mAP	$AP_{Embedded}$	$AP_{Isolated}$
#1	**0.98248**	0.14826	0.25764	0.90167	0.90436	0.89899
#2	0.96232	0.15936	0.27345	0.88445	0.89755	0.87135
#3	0.97619	0.13007	0.22956	0.89725	0.89141	0.90309
#4	0.97380	0.13758	0.24110	0.89732	0.90034	0.89430
#5	0.96866	0.03595	0.06933	0.89934	0.89945	0.89922
#6	0.60120	**0.55002**	**0.57447**	0.61916	0.36002	0.87830
#7	0.98795	0.01734	0.03408	0.45424	0.00000	0.90849
#8	0.97142	0.03595	0.06934	0.89981	0.89945	0.90016
#9	**1.00000**	0.00835	0.01657	0.50000	0.00000	**1.00000**
#10	0.913242	0.02115	0.04134	**0.94903**	**1.00000**	0.89806

Table 4.1: **Results of SSD**

Trial	Backbone	Max. Resolution	Pre-trained	Num. priors	Parameters	Time to train
#1	VGG16	1024x1024	No	7020	stride=2; loc_weight=1.0; nhm=3:1	8h 21min
#2	VGG16	1024x1024	No	7020	stride=2; loc_weight=2.0; nhm=3:1	8h 17min
#3	VGG16	1024x1024	No	7020	stride=2; loc_weight=1.0; nhm=2:1	8h 22min
#4	VGG16	1024x1024	No	12512	stride=2; loc_weight=1.0; nhm=2:1	8h 22min
#5	VGG16	1024x1024	No	34272	loc_weight=1.0; nhm=1:1	9h 10min[4]
#6	VGG16	1024x1024	No	34272	loc_weight=2.0; nhm=2:1	11h 40min
#7	VGG16	1024x1024	No	34272	loc_weight=1.0; nhm=3:1	11h 39min
#8	VGG16	1024x1024	No	34272	loc_weight=2.0; nhm=3:1	9h 8min[5]
#9	VGG16	800x1024	No	26788	loc_weight=1.0; nhm=3:1	9h 45min
#10	VGG16	800x1024	No	57012	loc_weight=1.0; nhm=3:1; 600epochs	19h 39min

Table 4.2: **Experiments of SSD**

Model analysis

From Table 4.1 we see that despite having a high mean average precision in all trials, the recall is usually quite low except for trial #6 (where it is still low, but on the edge of usefulness). This recall issue is probably due to the constraints we had to enforce on our model (discussed in the previous section) and some bad parametrization.

First, we studied the effects of the localization weight in the results. Initially, we thought, there would be localization problems and this parameter should be increased. But later, the opposite happened. The initial localization was pretty good as long as the equation was found (recall problems). When the localization weight is increased, the recall tends to increase in exchange for a decrease in precision.

Then, we test the hard negative mining ratio from 3:1 to 2:1 with different localization weights. As the model is quite sensible in terms of parametrization, it is no clear how the hard negative mining really affects their results, but it seems that an increase in the hard negative mining leads to a decrease in the precision but an increase in the recall. Anyway, these results are not conclusive due to the poor recall values.

Then we train the models for more epochs to see in minor improvements in the loss had a big impact on the results but the impact turned out to be minimal. More epochs did increase the f1-score but marginally.

In addition to this, we tested different image resolutions for efficiency reasons (it decreases the number of priors). Using fewer priors for the exact same network and same parameters led to a minor improvement in the results. This is probably because the model now only trains using the priors that are really susceptible to be used. Previously, the

useless ones were ignored by the model through training, but now they are not included from the beginning.

At the same time we were performing these experiments, we tested different methods for generating priors as well as the behavior of the model when the number of prior was changed. From the results, increasing the number of priors seem that only harms the model but this is simply the effects of collateral problems derived from the methods used when choosing these priors. Although not documented here, we did other small tests where we found out that increasing the number of priors lead to improvements in the results but in exchange for increasing the training times.

The results in this section are bad. They are not really useful from an application standpoint. By showing these results we wanted to record some of the experiments we have performed. However, due to the time limitation for this work, we couldn't achieve what we wanted, an SSD-like model that uses a low number of high-quality priors to detect mathematical expressions. A natural chapter in the course of research is to fail. Not failing means that we are not trying new things, and incremental improvements or minor tunings in the original model were not our focus. From other experiments, using a standard parametrization and lots of priors such as the ones from the original paper will lead to good results.

Finally, we have commented earlier in this section that we pre-trained SSD[4] on the Pascal VOC[2] dataset[6]. The main benefit we had from performing transfer learning is that the initial confidence was greater. This led to fewer final hypothesis, which meant a smaller IoU matrix and resulted in no more memory-related problems for us. Later, we did more modifications in the architecture and we could mitigate these problems without performing transfer learning.

In the Figure 4.6, we have four samples (pages) with the model predictions and their corresponding ground-truths. (To get these predictions, the non-maximum-suppression ratio was set to 0.3; depending on its value results may vary).

[6]This experiment is not in the results table since the final results on the marmot dataset were not good, and the experimentation was not robust and did not contribute to getting any numerical insight

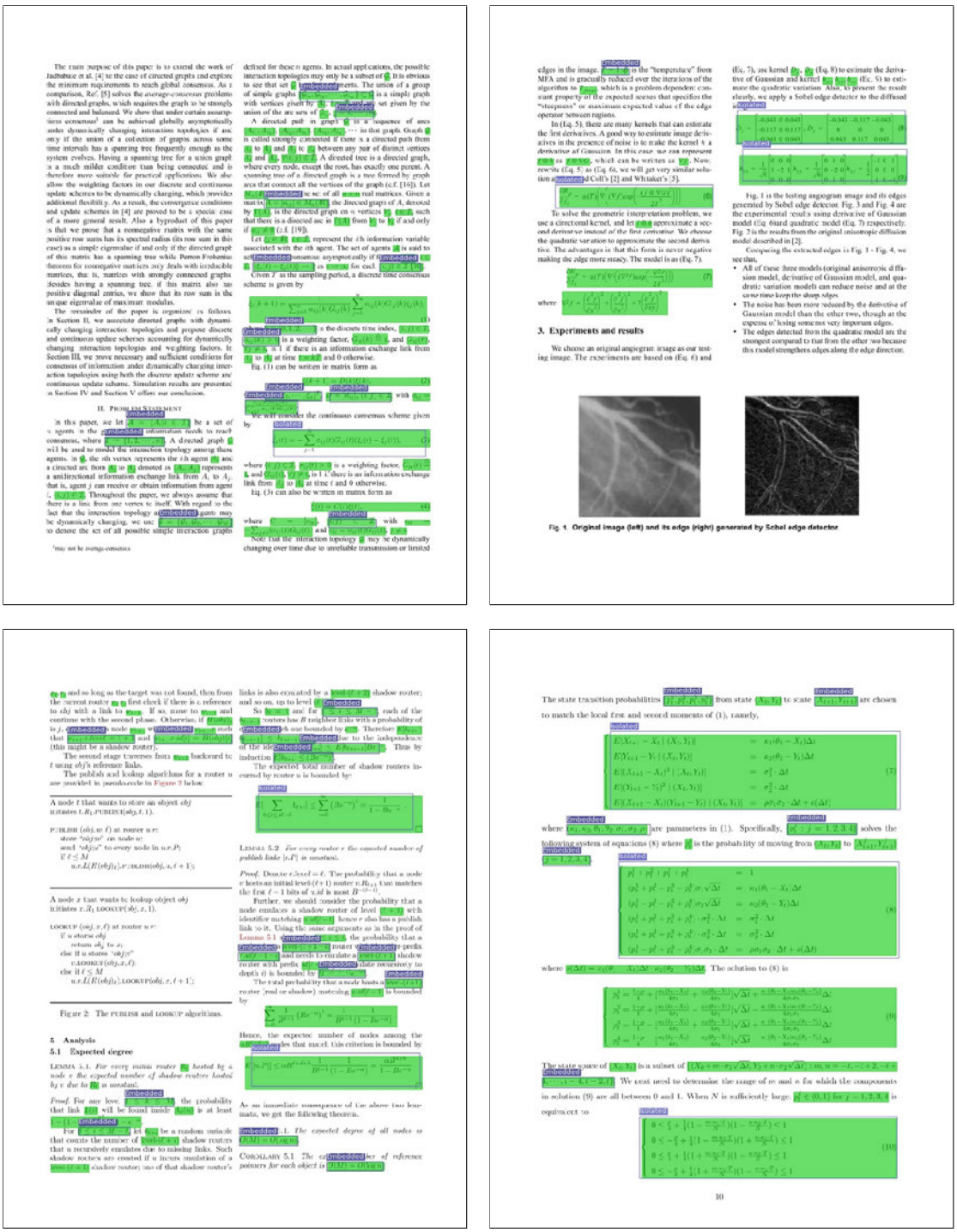


Figure 4.6: **Results of SSD:** In green the ground-truths and in blue the predicted bounding boxes

As we guessed from the numerical results, the recall is notably low although the models seem to be quite precise. As discussed earlier, this issue should be relatively easy to fix by applying some minor modifications. Again, due to the time limitations and the time spent on improving the original architecture for this task, we couldn't get good results on time.

To further examine the performance of SSD[4] we have breakdown the results using the methodology proposed by Hoiem et al.[16] as discussed earlier. It is important to

point out that due to the poor recall values, theses results present are strongly biased. Nevertheless, the serve as a baseline for future models or improvements.

To analyze the errors of our model, for each class at test time we have taken the top-N predictions. Then, each prediction is either correct or incorrect but with a specific type of error (localization, other and background).

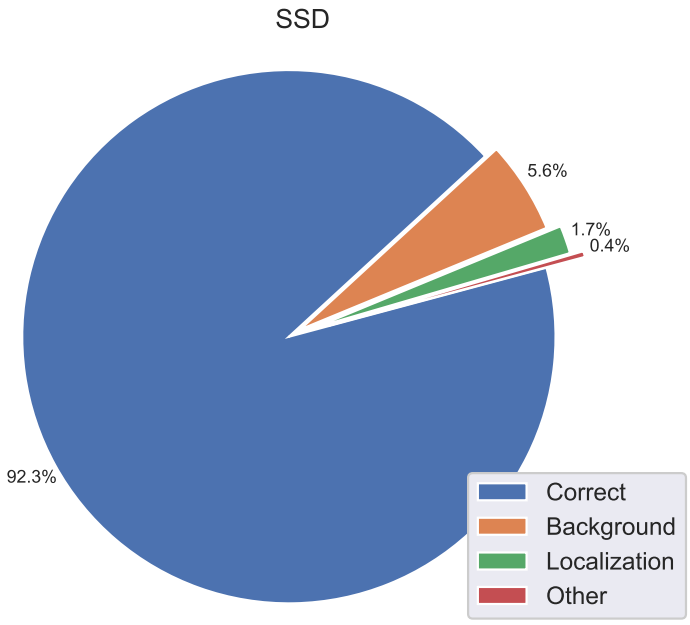


Figure 4.7: **Error analysis of SSD**: This chart shows the percentage of type errors in the top N detections for different classes

From Figure 4.7 we see that SSD[4] has an impressive rate of correct detections (92.3%) but it struggles the most with the background errors were the model mistakes it as an expression 5.6% of the times. The localization errors only account for the 1.7% of the detections while the wrong detections are minimal, just the 0.4% of the total.

Nevertheless, as we have said before, the main problem of our model is its low recall value. It is so low that the usefulness of the model is anecdotal despite having a high precision. Anyway, using more memory, a better set of anchor boxes, some minor architecture changes and a more careful tuning to deal with the loss weights and the hard negative mining ratio will probably lead to a substantial improvement of this model.

Then, we have plotted the Recall-Precision curve. This curve tells us how good is the classifier when classes are very imbalanced (as it's our case). From Figure 4.8 we can see than our model behaves quite well with an impressive average precision for both classes.

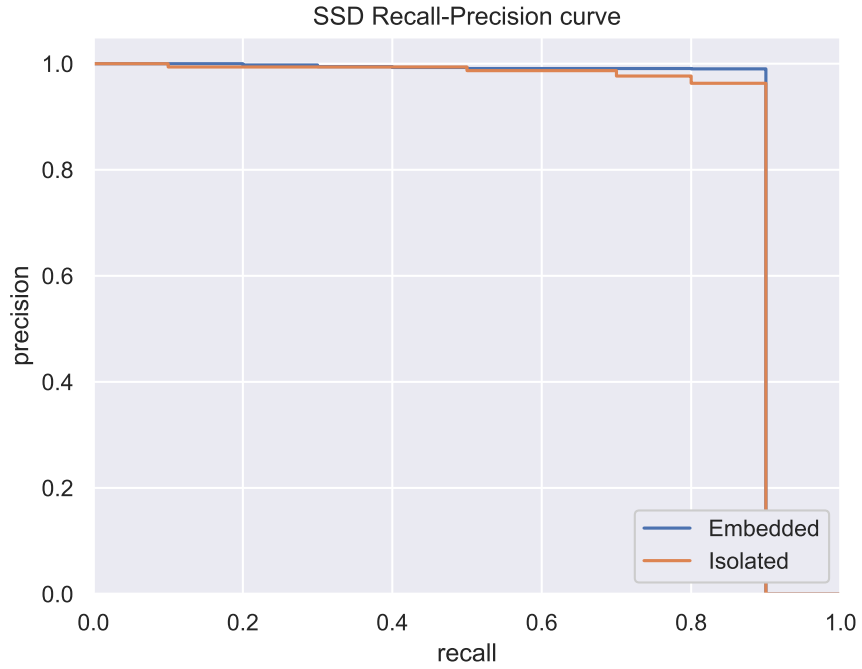


Figure 4.8: **Recall-Precision curve of SSD**

Finally, we have computed the ROC curve to understand the performance of the classification part of our detector, at different thresholds for each class. From Figure 4.9 we see that the model has less problems classifying isolated expressions than the embedded ones. This is expected as embedded expressions can be easily mistaken as normal text.

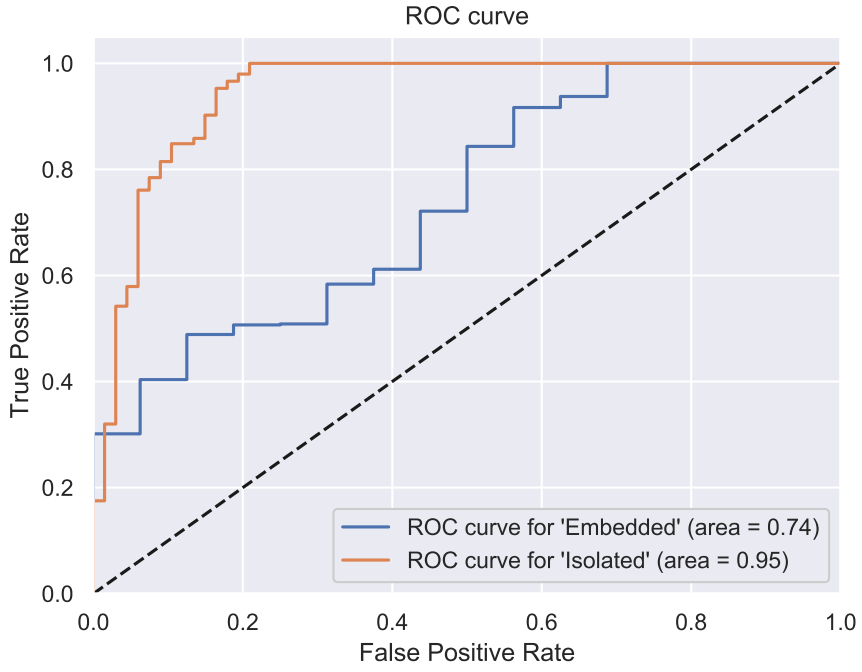


Figure 4.9: **ROC curve of SSD**

Again, the results from Figures 4.8 and 4.9 should be taken with a grain of salt due to the low recall value. In a future work, we will address the issues that due to the time limitations we couldn't solve on time.

4.2.2 YOLO

Network design

YOLOv3[15] introduces a state-of-the-art architecture which trades-off accuracy and speed depending on the input resolution. In the paper, the authors present three input resolutions: 320x320, 416x416, and 608x608. We decided to test the detector with the highest accuracy (*YOLOv3-608*) and a less accurate but highly efficient version of the 416x416 network, called *YOLOv3-tiny*.

The first model, *YOLOv3-608*, uses *Darknet-53* as the backbone for feature extraction. It has been trained and tested on the COCO dataset, has 140.69 billion parameters and runs at 20FPS with mAP of 57.9.

The second model, *YOLOv3-tiny*, uses *Darknet-Tiny*, a small but highly efficient feature extractor for constrained environments. Similarly, it has been trained and tested on the COCO dataset, but has 33.1 billion parameters and runs at a 220FPS with mAP of 33.1.

To meet the requirements of our task (mathematical expression detection), we increased the input resolution of the model to a minimum of 1024 pixels so that there was enough room to detect character-level details.

In addition to this, there have been minor modifications in the architecture to adapt it to the number of clusters used in each experiment as well as the number of classes used.

Finding clusters

Before training the model, we have to choose the anchors that will be used as priors. These anchors are domain-dependent so instead of choosing them by hand or running some kind of heuristic or grid, we simply run K-means. This allows us to get good priors for the specific dataset we're gonna use. The exact process has been described in previous sections.

We did not want to choose an arbitrary number of clusters so we run K-mean from 1 to 9 clusters to know how the average IoU varies along with the number of clusters.

From the Figure 4.10 we see that the more clusters we use, the higher the average IoU is. Although this might lead us to think that using more clusters will lead to better accuracy, in practice that is not what happens as there are more factors to take into account such as the ability of the model to deal with a large number of clusters. Furthermore, the biggest increase in the average IoU is produced when stepping from cluster 1 to 2. A plausible explanation for this leap could be that we only have two classes of objects, with both having different dimensions (embedded-small vs. isolated-big).

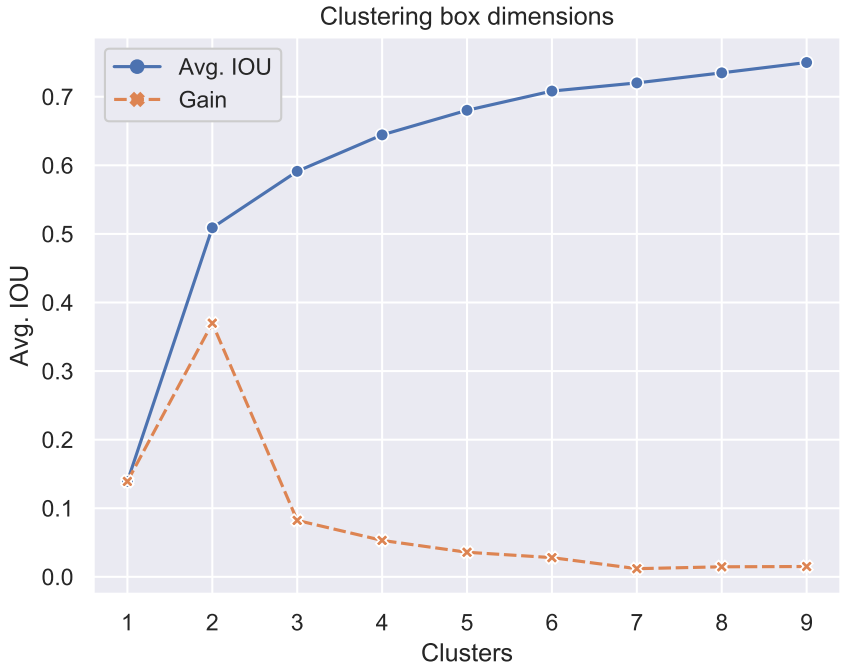


Figure 4.10: **Clustering box dimensions for Marmot**

In the results section, we will see the relationship between the number of clusters and accuracy, and how the rule of *the more, the better* does not necessarily apply.

Training

Each YOLO model that we have trained have their own set of particularities and settings tailored for the experiment we were running. But generally speaking, we have followed a similar approach for all of them. Full images as the input preserving the aspect ratio (adding padding if necessary), Adam[32] as the stochastic function optimizer with a learning rate of 1e-3, a maximum of 300 epochs or 20h of training, 90% of the training data for training and 10% for validation, small batch sizes ranging from 1 to 8 due to memory limitations in our GPU, 1 to 2 gradient accumulations per optimization step, no hard negative mining and a relatively standard data augmentation.

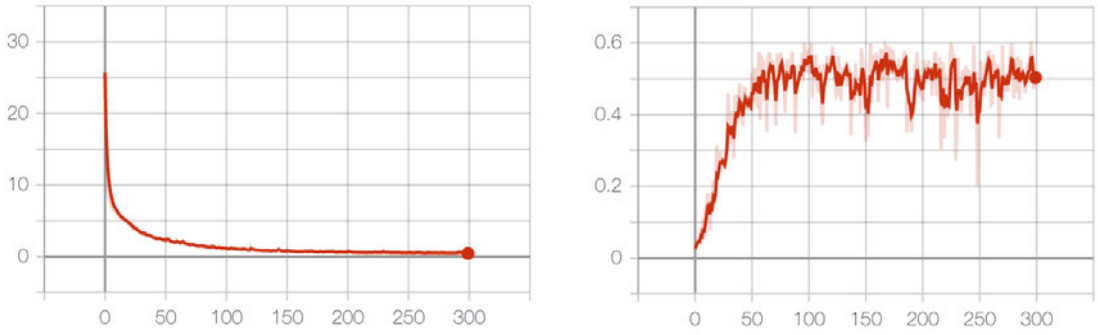
For the data augmentation, we used the following transforms:[7]

- **Shift:** $[-0.0625, +0.0625]$
- **Scale:** $[-0.0, +0.0625]$
- **Rotate:** $[-2, +2]$
- **Interpolation:** Bilinear interpolation with coefficients (1, 0)[8]
- **Border mode:** Replicate
- **Color:** RGB (3-channel) or Gray (1-channel)
- **Resize mode:** Keep aspect ratio

[7] Some of them might vary depending on the experiment.
[8] *INTER_AREA* from OpenCV

- **Longest max. size:** 1024, 1280, 1440

As the YOLO's loss function is made up of multiple components, we had to monitor each one individually to be sure that the model is correctly implemented. This aspect will be briefly discussed in a later section since it's more implementation-related than relevant for the research itself. For now, the most relevant thing is to know the smoothness of the overall loss function, the recall, precision, and f1-score. See Figure 4.11.



(a) Total loss (b) mAP

Figure 4.11: **Total loss and mAP of YOLOv3**

Results

The results showed in Table 4.3 are computed using an IoU of 0.5, a confidence threshold of 0.5, a non-maximum suppression of 0.3 and the mean average precision has been computed over all the points and 11 points to compare.[9]

These results will be discussed in the next section.[10]

Trial	Precision	Recall	F1	mAP	$AP_{Embedded}$	$AP_{Isolated}$
#1	0.72297	0.88379	0.79530	0.78485	0.74852	0.82118
#2	0.86098	0.92744	0.89202	0.85413	0.82208	0.88618
#3	0.82002	0.88961	0.85287	0.81357	0.79125	0.83589
#4	0.87030	0.86343	0.86681	0.80369	0.73157	0.87580
#5	0.88594	0.91716	**0.90092**	**0.87434**	**0.87770**	0.87099
#6	0.78895	0.64256	0.70581	0.55846	0.56713	0.54978
#7	0.89594	0.86077	0.87768	0.81209	0.72352	**0.90065**
#8	0.44704	**0.95334**	0.60014	0.74550	0.74397	0.74703
#9	**0.90124**	0.59835	0.71920	0.56173	0.54023	0.58324
#10	0.78506	0.79954	0.79207	0.67387	0.61303	0.73472
#2 (11-point)	0.81264	0.91782	0.86203	0.75836	0.65852	0.85821
#8 (11-point)	0.36028	0.94849	0.52221	0.36851	0.29190	0.44511

Table 4.3: **Results of YOLOv3**

[9] We suspect that there could be some small error in the implementation that due to lack of time it has been not possible to review it previous to this submission. In any case, since all models have been evaluated in the same way, using the same function evaluation functions, the results should be valid for comparison between them as they all would have the same bias.

[10] C = Num. of clusters; loc = localization weight; obj/noobj = confidence weights

Trial	Backbone	Max. Resolution	Pre-trained	Parameters	Multi-scale	Time to train
#1	Darknet-53	1024x1024	No	3C; loc=1.0; obj=1.0; noobj=100.0	No	11h 56min
#2	Darknet-53	1024x1024	No	6C; loc=1.0; obj=1.0; noobj=100.0	No	11h 55min
#3	Darknet-53	1024x1024	No	9C; loc=1.0; obj=1.0; noobj=100.0	No	11h 54min
#4	Darknet-53	1024x1024	Yes	6C; loc=1.0; obj=1.0; noobj=100.0	No	11h 59min
#5	Darknet-53	1280x1280	No	6C; loc=1.0; obj=1.0; noobj=100.0	No	17h 36min
#6	Darknet-Tiny	1024x1024	No	6C; loc=1.0; obj=1.0; noobj=100.0	No	2h 28min
#7	Darknet-53	1024x1024	No	6C; loc=1.0; obj=1.0; noobj=100.0	Yes	12h 31min
#8	Darknet-53	1024x1024	No	6C; loc=5.0; obj=1.0; noobj=0.5	No	9h 13min[11]
#9 - FT#2 50epochs	Darknet-53	1024x1024	No	6C; loc=10.0; obj=1.0; noobj=100	No	1h 30min[12]
#10 - FT#2 50epochs	Darknet-53	1024x1024	No	6C; loc=1.0; obj=1.0; noobj=50	No	1h 35min[13]

Table 4.4: **Expermients of YOLOv3**

Model analysis

From Table 4.3 we can extract that we need a strong backbone capable of extracting the detail needed for this task. Here, Darknet-53 showed the best performance but it would be interesting to try something different such as VGG[17] or ResNet[18]. This would substantially decrease the detector speed but depending on the case, a real-time detector might be not needed, but an accurate one.

Then, we tried different image resolutions: 512, 1024 and 1280. We didn't show the results on the 512-detector because it wasn't enough to capture many of the character-level embedded expressions. Between the 1024- and 1280-detector, the latter was the winner. Training with high-resolution images improved most of the detection metrics. This was expected since detectors, in general, need high-resolution images to get good detection results. However, the main problem when using high-resolution images is that although they improve the accuracy of the model, they tend to notably increase the training times.

Since we wanted to test the original Darknet[33] YOLOv3-608 weights in order to start the training from a pre-trained network, we had to write a special function to read the weights of each layer of the original file, and then copy them to the layers of our model in a specific order. Easier said than done. Anyway, once we did it and check that the model worked correctly, we trained our model using those weights. Interestingly, and against all odds, using pre-trained weights did not improve the accuracy of our model (actually there was a slight increase in the precision but in exchange of losing 5 percentage points in the mean average precision).

After that, we study the impact of the number of clusters in the accuracy of the model. We found out that between 3, 6 and 9 clusters, 6 was the right amount. When 6 clusters were used, we got the highest mAP with an impressive ~7% increase with respect to the model that used 3 clusters and a~4% for the one with 9 clusters.

In the original paper, multi-scale training improved the accuracy of the model, but here, instead of improving it, multi-scale training made the model less accurate. This might be because most of the mathematical expressions found in our dataset have practically the same scale and most differences are due to the minor data augmentation that it is applied.

Finally, we cannot forget to perform a visual inspection comparing the predictions with the ground truth. In the Figure 4.12, we have four samples (pages) with the model predictions and their corresponding ground-truths. (To get these predictions, the non-maximum-suppression ratio was set to 0.5; depending on its value results may vary).

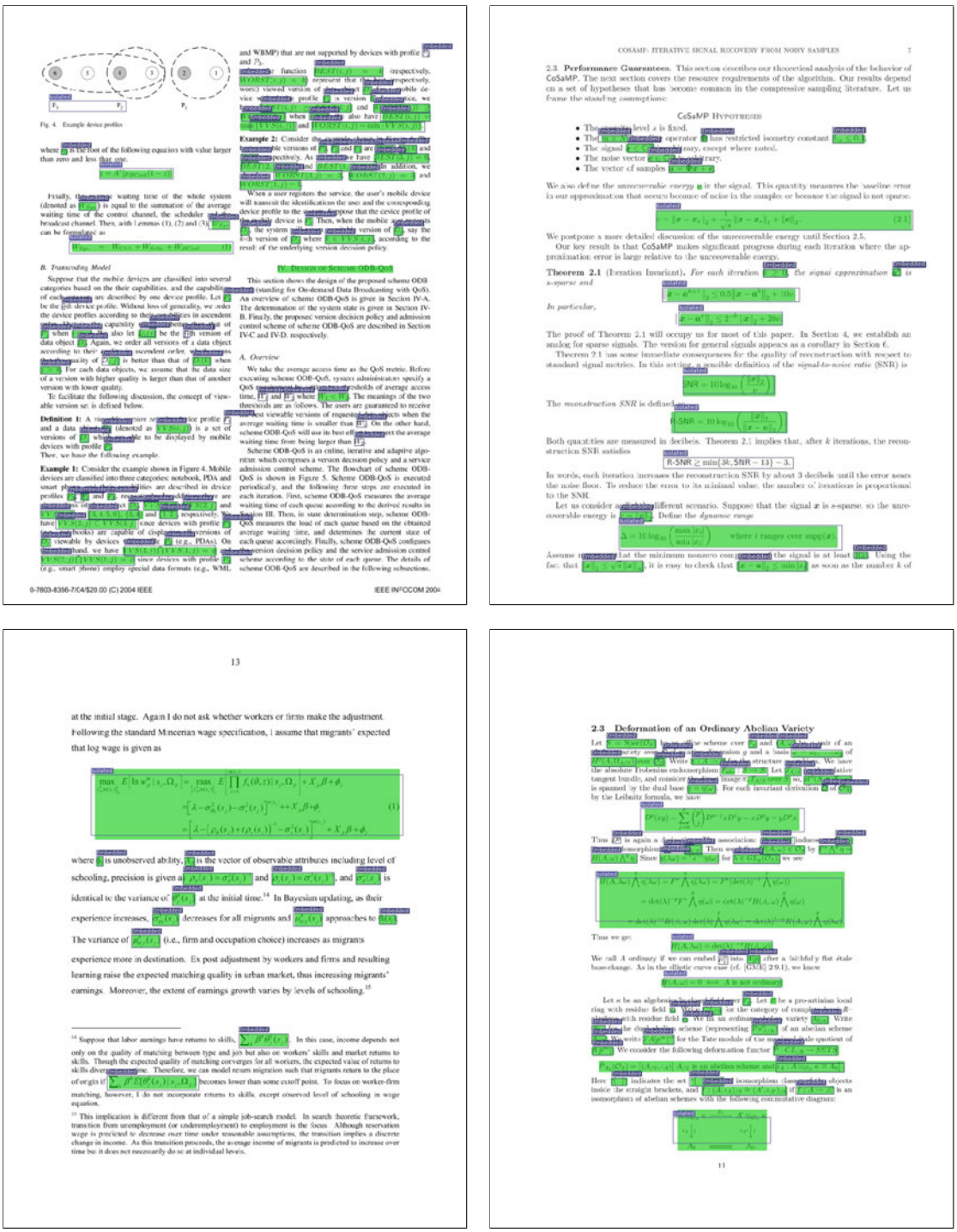


Figure 4.12: **Results of YOLOv3**: In green the ground-truths and in blue the predicted bounding boxes

The first thing we noticed is that the model is pretty robust in terms of classification but there are problems fitting the bounding boxes to the ground-truths. Besides, the model seems to fail when equations come with unusual dimensions. An example of this would be a mathematical expression that is taller than wider such as a matrix, or multi-line expressions. To fix this we have to give a higher weight to the *coordinate constant* in the loss function, and then we could work on getting better priors and do a more careful tuning. Following these steps will likely improve in a meaningful way the results of this model.

As we have done with SSD[4], to examine from a different perspective the performance of the model, we have breakdown the results using the methodology proposed by Hoiem et al.[16] (see Figure 4.13). Again, for each class at test time, we take the top N predictions, and then, each prediction is either correct or incorrect but with a specific type of error (discussed earlier).

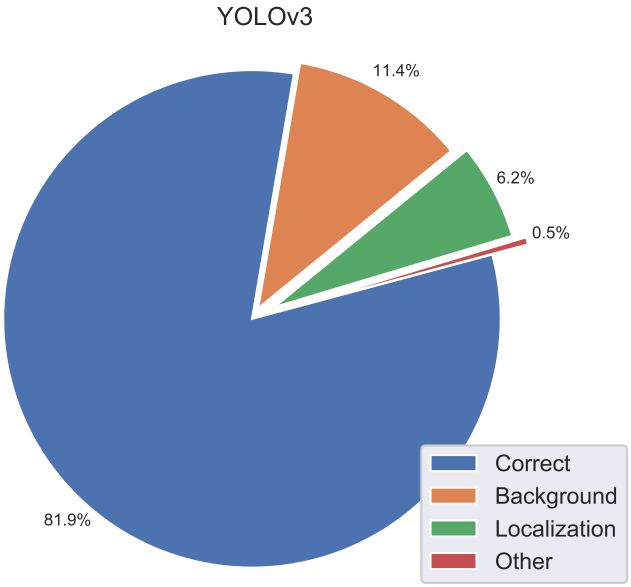


Figure 4.13: **Error analysis of YOLOv3**: This chart shows the percentage of type errors in the top N detections for different classes

In this model, YOLO struggles more with background errors and localization errors than miss-classifications. However, it is pretty balanced in terms of accuracy and recall. From Figure 4.13 we see that YOLO has an acceptable rate of correct detections (81.9%) but it struggles with the background errors were the model mistakes it as an expression 11.4% of the times. The localization errors only account for the 6.2% of the detections while the wrong detections are minimal, just the 0.5% of the total.

Once again, we have plotted the Recall-Precision curve. From Figure 4.14 we can see than our model behaves quite well until we increase the recall of the embedded expressions up to 0.7 and up to 0.9 for the isolated ones.

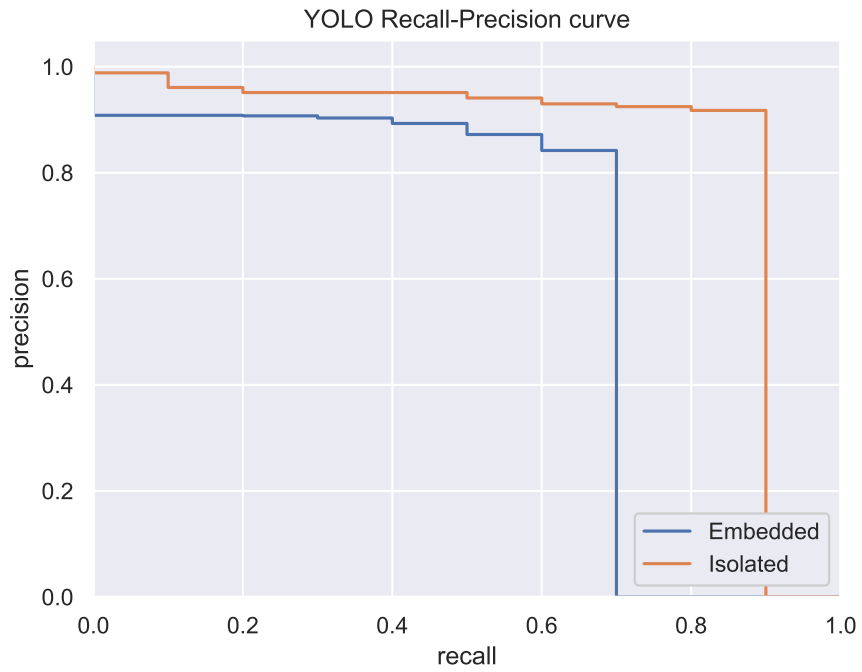


Figure 4.14: **Recall-Precision curve of YOLOv3**

Finally, we have computed the ROC curve to understand the performance of the classification part of our detector, at different thresholds for each class. From Figure 4.15 we see that the model has less problems classifying isolated expressions than the embedded ones. Again, this is expected as embedded expressions can be easily mistaken as normal text.

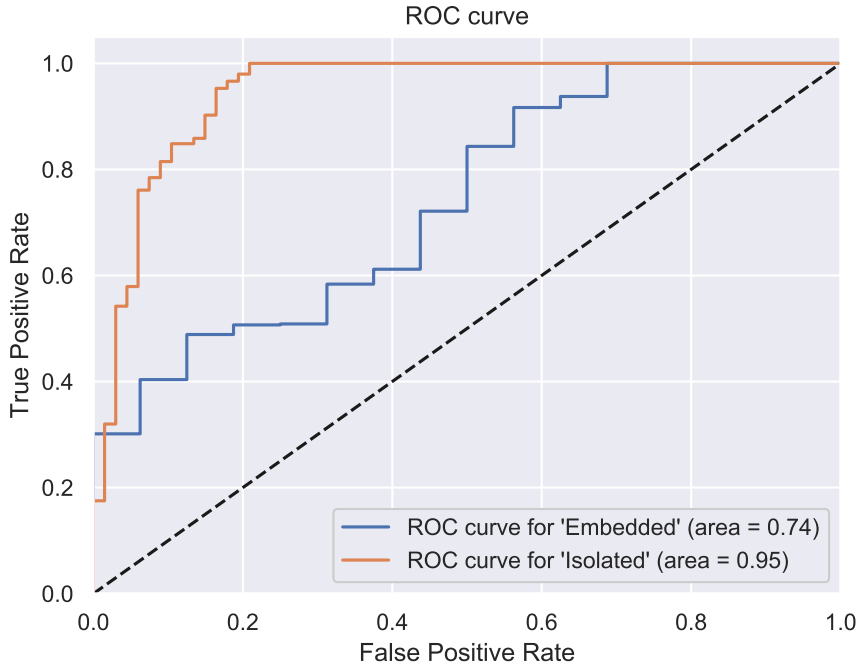


Figure 4.15: **ROC curve of YOLOv3**

4.2.3 Model debugging

The models presented in this work (SSD[4] and YOLOv3[15]) might be easy to understand from a conceptual level, but when one needs to implement them many problems may arise that are difficult to debug due to the black-box nature of neural networks.

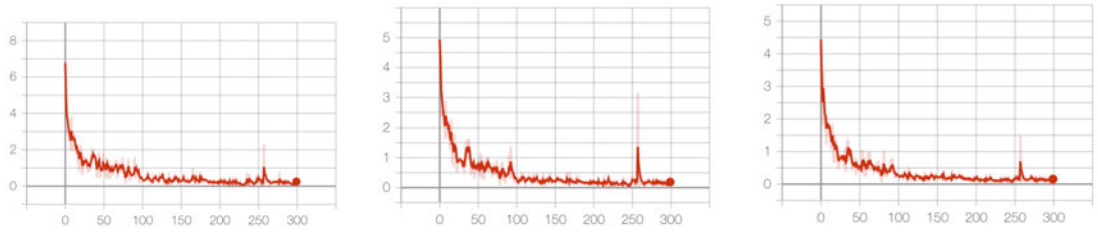
When a model is not working as expected we need to understand what it is exactly learning. To do this, we can check the loss function during training looking if it is stable, converging, diverging, or a combination of them in multi-loss functions. If this is the case, we have to plot all the individual components of the objective functions so that we can easily see which parameters are being optimized correctly and which don't.

Furthermore, it is interesting to plot not only the objective function or its parts, but also the metrics that are relevant in order to avoid overfitting or to be able to make a better diagnosis of the weaknesses of our model.

As we are dealing with an object detection task, our main metric will be the mean average precision. A good sign that our model is not being overfitted is that the training loss, the validation loss, and here, the mean average precision as well as other metrics, are strongly correlated.

In this section we will only analyze the training process of YOLOv3[15], since it is practically the same for SSD[4] and would not contribute at all to repeat the same thing twice.

Let's start with the total loss function of each prediction layer. In the Figure 4.16 all three layers seem to converge to a minimum value and then, they remain stable. This means that our model is not learning anymore which might be a symptom that we need to decrease the learning rate, or if the validation loss and the training loss diverge might indicate a possible overfit, or simply that the model has reached its learning capacity.

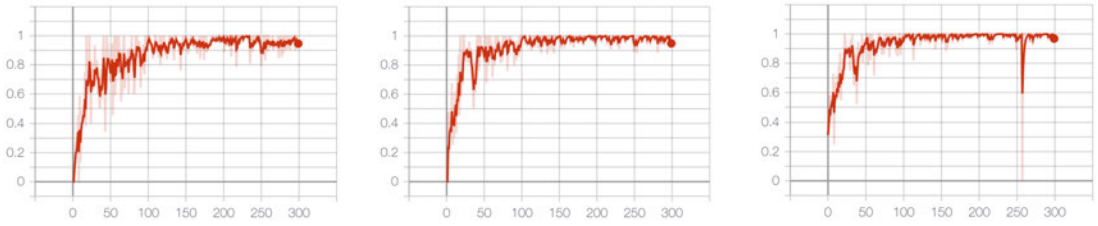


(a) Loss 1 (Small objects) (b) Loss 2 (Medium objects) (c) Loss 3 (Big objects)

Figure 4.16: **Multi-part loss**

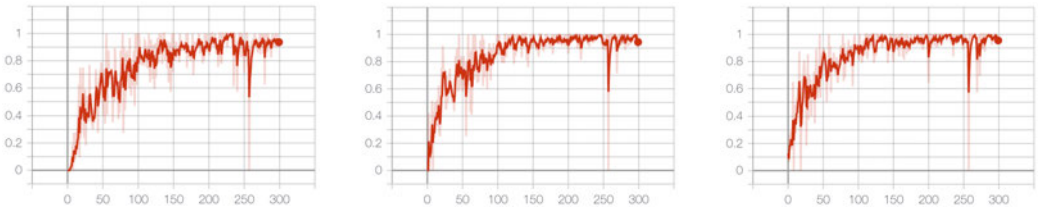
Along with the total loss, it is strongly advisable to plot the rest of the metrics such as the recall, precision, f1-score or mean average precision (in this particular case).

In a previous section has been commented that one of the problems of these metrics is that they vary depending on the IoU threshold. To account for this issue, we have plotted different versions of them, one for an IoU of 0.5 and another for an IoU of 0.75. See Figures 4.17 and 4.18 for the recall and Figure 4.19 for the precision.



(a) Recall50 (Small objects) (b) Recall50 (Medium objects) (c) Recall50 (Big objects)

Figure 4.17: **Recall with an IoU of 0.5**

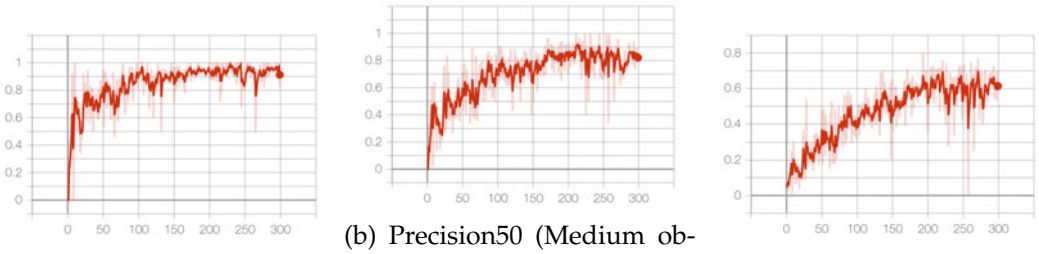


(a) Recall75 (Small objects) (b) Recall75 (Medium objects) (c) Recall75 (Big objects)

Figure 4.18: **Recall with an IoU of 0.75**

In the precision figures 4.19 we can see something interesting. In the past, YOLOv1[3] and YOLOv2[13] used to struggle with small objects but when YOLOv3[15] came to the scene, the trend somehow turned around. Now YOLO shows a remarkably good performance on small objects and comparatively worse on medium and large objects.

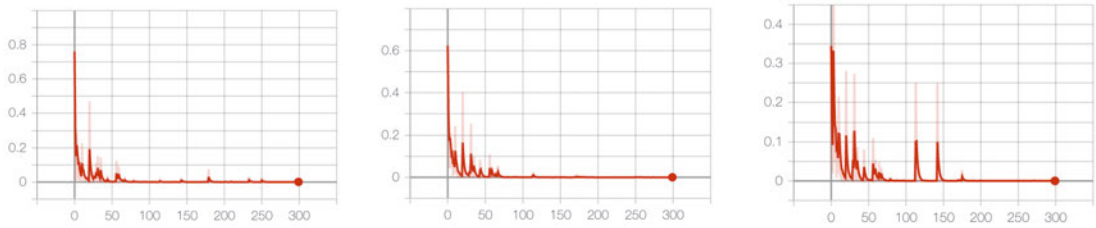
This effect can clearly be seen in the third Figure 4.19c, where the precision not only takes more iterations to converge but also, the final ratio is notably lower than in the charts from the small and medium objects. Hence, we should work on that aspect to push the accuracy of our model further.



(a) Precision50 (Small objects) (b) Precision50 (Medium objects) (c) Precision50 (Big objects)

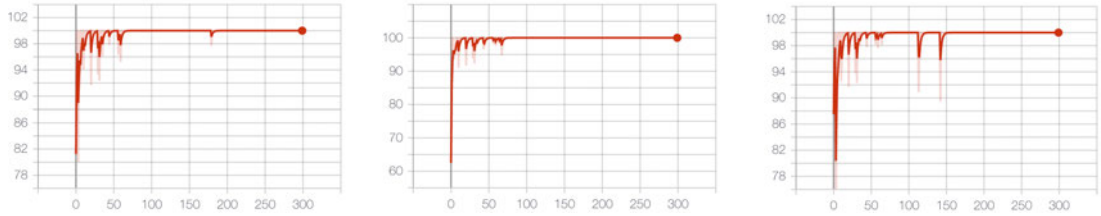
Figure 4.19: **Precision with an IoU of 0.5**

Now we look at the classification loss and classification accuracy (see Figure 4.20 and 4.21). Each prediction layer is specialized in one size of objects (big, medium and big). As it has been commented early, the classification loss almost zero and the accuracy around 100% in all three classifiers which means that it's pretty easy for our model to distinguish between embedded and isolated math expressions. This is not surprising since isolated expressions are bigger and they are also separated from the text by a substantial white margin which is easy to detect by the model, while the embedded expressions are smaller and can easily go unnoticed since they are embedded in the document body.



(a) Cls 1 (Small objects) (b) Cls 2 (Medium objects) (c) Cls 3 (Big objects)

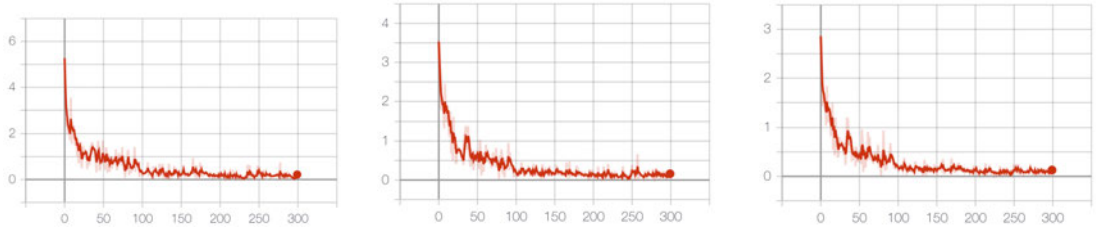
Figure 4.20: **Classifier loss**



(a) Cls acc. (Small objects) (b) Cls acc. (Medium objects) (c) Cls acc. (Big objects)

Figure 4.21: **Classifier accuracy**

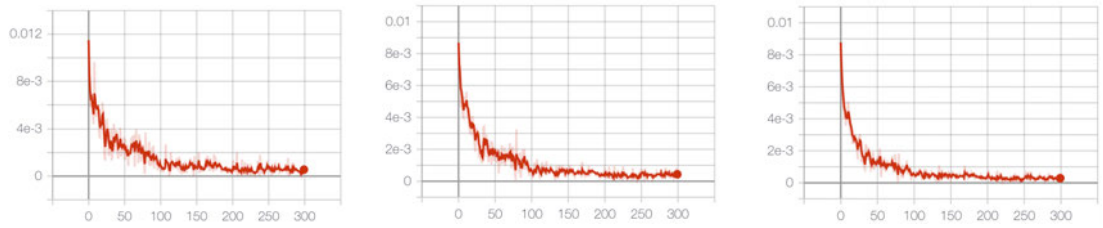
In the next figures (4.22, 4.23, 4.24) we will study the confidence of our model in distinguishing the background from relevant objects. Similar to the previous figures, the confidence loss decreases as it should in all three models. This loss is made up of the *no-object confidence loss* and the *object confidence loss*. Hence, if both decrease, the overall confidence loss should decrease too.



(a) Conf. (Small objects) (b) Conf. (Medium objects) (c) Conf. (Big objects)

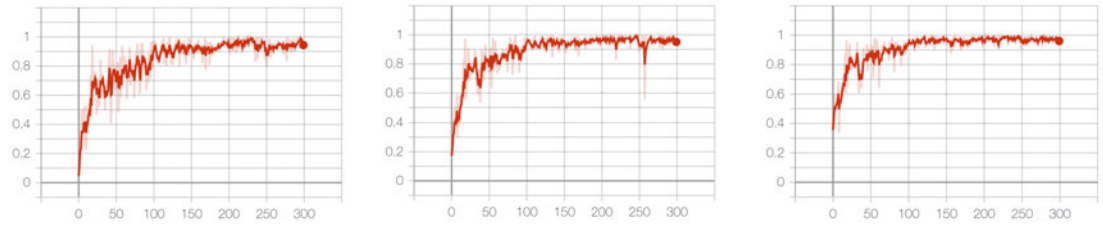
Figure 4.22: **Confidence loss**

It is important to remark that in Figures 4.23 and 4.24 we see the confidence as a probability, not the loss. In the Figure 4.23 we see something interesting. The values on that chart should be close to 1 meaning that the model is confident when it predicts the background. Nevertheless, the opposite happens. That is, the model has almost no confidence when it predicts something as background. This is because many grids cells (YOLO) do not contain any object and the optimizer reacts by pushing the confidence of those cells towards zero overpowering the gradient of cells that do contain the object. To remedy this, we should decrease the loss confidence for predictions that don't contain any object.



(a) No-obj conf. (Small objs) (b) No-obj conf. (Medium objs) (c) No-obj conf. (Big objs)

Figure 4.23: **No-object confidence**

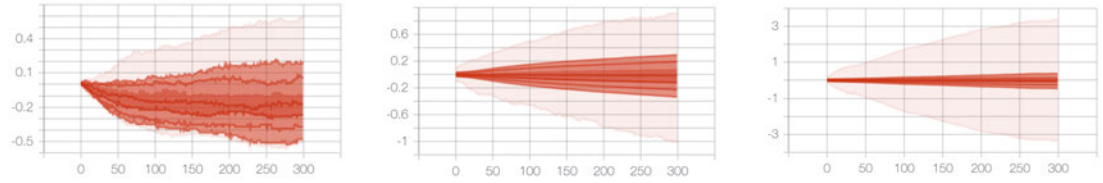


(a) Obj. conf. (Small objects) (b) Obj. conf. (Medium objects) (c) Obj. conf. 3 (Big objects)

Figure 4.24: **Object confidence**

Finally, it is recommended to plot the distributions and histograms of weights and biases to double-check that everything it's being optimized correctly or to diagnose redundant layers.

For instance in Figure 4.25 we can see the weights and bias distributions from three different layers. At first sight, they look completely fine but usually, we should look for abnormal values or values that do not evolve with the training. A value that does not change might indicate that there could be something wrong with the model.

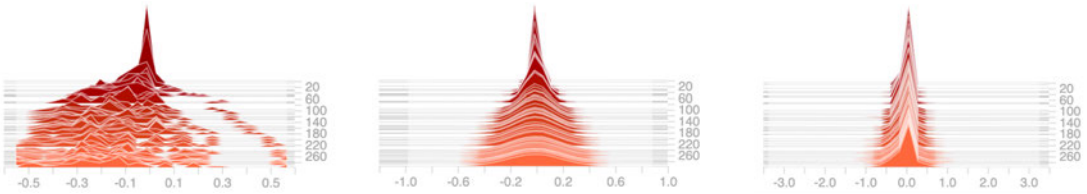


(a) Bias distr. BatchNorm0 (b) Weight distr. Conv10 (c) Weight distr. Conv105

Figure 4.25: **Weight and Bias distributions**

It is important to remark that if a distribution is completely flat or abnormal to us, doesn't necessarily mean that there is something wrong with our model. Although we should take a closer look and understand why this is happening. It could be due to a bug, a numeric instability problem or it's simply a normal behavior in that specific context.

Similarly, we can take a look at the histograms of each layer to see the evolution of weights as biases from a different perspective. See Figure 4.26.



(a) Bias hist. BatchNorm0 (b) Weight hist. Conv10 (c) Weight hist. Conv105

Figure 4.26: **Weight and Bias histograms**

4.3 Model calibration

The results of a trained model might vary depending on the evaluation parameters.

To understand how our model behaves under different parameters, we have re-evaluated every model using a grid of values ranging from 0.1 to 0.9 with a step of 0.3 for the IOU, confidence, non-maximum suppression thresholds.

First, let's start with the IoU threshold. This threshold says how much a correct prediction needs to overlap with the ground-truth to be considered as *positive*. The higher we set the threshold, the fewer predictions will pass as positives. This means that increasing the IoU threshold will decrease the values of the metrics as we are setting a high standard for what we consider a positive prediction. Indeed, this effect can clearly be seen in Figure 4.27.

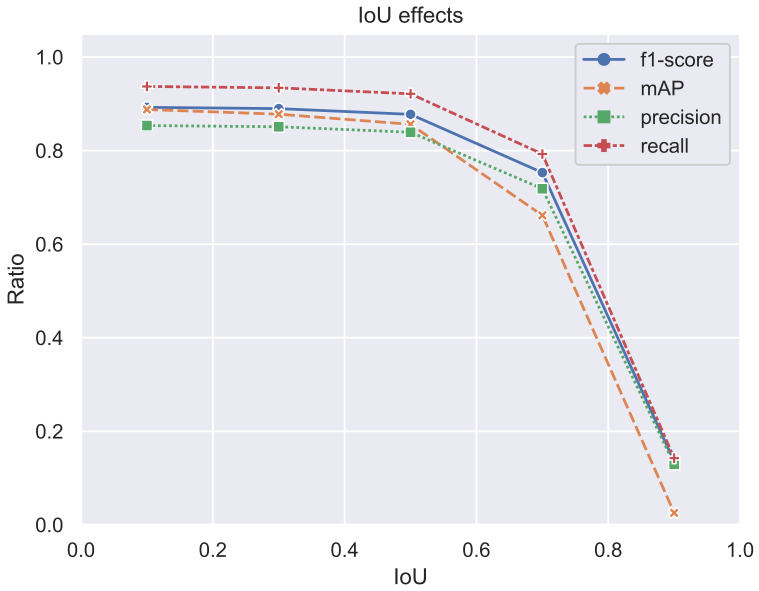


Figure 4.27: **IoU effects on the metrics**

Now, moving towards the confidence threshold in YOLO's model, we start to see interesting things. The confidence value allows us to keep only the predictions the model is sure of. In theory, a higher confidence threshold should increase the precision of a model and decrease its recall. However, we do not see that from the curves in Figure 4.28. This is the result of the abnormality that has been discussed in the previous section. A poorly balanced loss function or more specifically, a poorly balanced confidence loss leads to overpower the gradients of cells than do contain an object. This forces those that

do not contain one to converge towards zero. Consequently, we can say that the chart from Figure 4.28 gives us no meaningful information (beyond of knowing that there is a problem with our model and needs to be corrected) so all confidence values are either 0 or 1, with almost no steps in between.

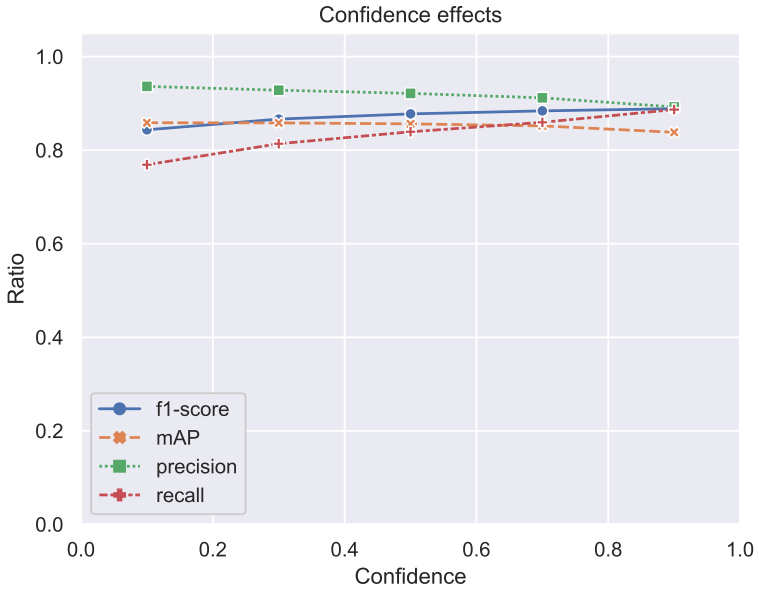


Figure 4.28: **Confidence effects on the metrics (Anomaly)**

Finally, the non-maximum suppression step. If we set a high threshold, the model will be very permissive with the amount of overlapping between predictions. This will improve the recall of our model as now there are more boxes, but it will harm other metrics such as the precision because there is more room for miss-classifications. Besides, many object detector competitions give a penalty for each *extra* or duplicate bounding box so this is something to take into account.

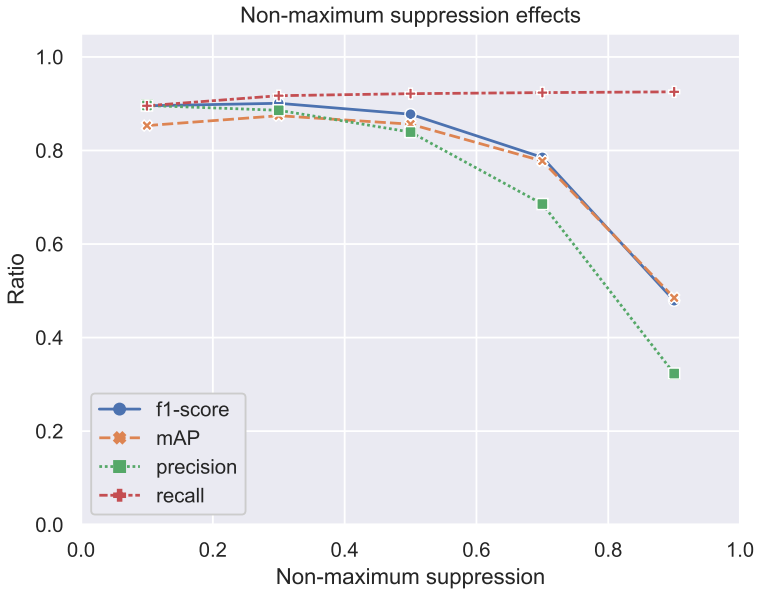


Figure 4.29: **NMS effects on the metrics**

4.4 Implementation notes

We used PyTorch[30] as the deep-learning library to build, train and evaluate our models. Everything is written in Python and the code can be found in Github[34].

All the models were trained and tested on a *NVIDIA GTX 1080 8GB* and a *NVIDIA GTX 1070 8GB*.

To further improve the speed of "slow" functions such as losses or metrics, all critical tensor operations were written in Pytorch[30] to make use of their efficient GPU support.

Chapter 5

Conclusions and future work

5.1 Conclusions

In this work, we addressed the problem of mathematical expression detection in images using single-shot object detectors. One of the main contributions of our work is to study the performance of two state-of-the-art models, SSD[4] and YOLOv3[15], for a not so common task.

A discussion on these models has been provided along with the singularities of the object detection evaluation. In particular, we have modified the architecture of SSD[4], YOLOv3[15], and the prior generation method to bring down the number of predictions into a manageable size. Some of the constraints imposed on the models were made to overcome the memory limitations we had, a thing that could have decreased the performance of these models. Our contribution here is two-fold. First an experimental comparison of their behavior has been performed, and second, we have provided a detailed analysis of their results along with some ideas on how to calibrate them.

The main focus of this work was to understand the idea behind state-of-the-art single-shot object detectors. SSD[4] has resulted quite difficult to optimize since minor changes on the prior generation or its parametrization lead to the stagnation of the model pretty easily. Nevertheless, it has shown an impressive high precision although with low recall values. On the other hand, YOLOv3[15] has shown remarkably good results with a balance recall-precision curve and a simple model parametrization.

Finally, we have studied the results of these models. As expected, both models have shown more problems to distinguish the embedded expressions from the isolated ones than the opposite. SSD[4] shows a correct prediction ratio of 92.3%, but it struggles the most with the background and the localization errors since they are predicted as expressions a 5.6% and 1.7% of the time, respectively. Meanwhile, YOLOv3[15] correctness ratio goes down to 81.9%, and the background and localization error rates go up to 11.4% and 6.2%.

5.2 Future work

Many experiments have been left for the future due to lack of time. Future work will seek for a deeper analysis of the two presented models, research of new architectures, better calibration, and more experimentation.

There are some ideas that I would have liked to try during the development of this work. This research has been mainly focused on two models and most of their experimentation has not been enough to push their limits. Hence, in the future the following ideas could be tested:

1. Random search[35] for a better parametrization.
2. State-of-the-art feature extractors such as ResNeXt[36], EfficientNet[37], SENet[38]
3. RAdam optimizer[39]
4. Different number of multi-scale auxiliary layers
5. Post-processing to optimize the fitness of each bounding box

Furthermore, other state-of-the-art object detectors could be tested (i.e. RetinaNet[40] or Faster R-CNN[12]). From the ideas of these works, we could design a new architecture to take advantage of the innovations that worked for mathematical expression detection and ignore those that did not contribute.

The initial work was focused on mathematical expression detection. However, we could take a different road and research the mathematical expression recognition problem.

Bibliography

[1] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[2] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–338, June 2010.

[3] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[5] M. C. Potter, B. Wyble, C. E. Hagmann, and E. S. McCourt, "Detecting meaning in rsvp at 13 ms per picture," *Attention, Perception, & Psychophysics*, vol. 76, pp. 270–279, Feb 2014.

[6] P. Felzenszwalb, D. McAllester, and D. Ramanan, "A discriminatively trained, multiscale, deformable part model," in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2008.

[7] J. R. Uijlings, K. E. Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *Int. J. Comput. Vision*, vol. 104, pp. 154–171, Sept. 2013.

[8] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov, "Scalable object detection using deep neural networks," *CoRR*, vol. abs/1312.2249, 2013.

[9] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. Lecun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *International Conference on Learning Representations (ICLR) (Banff)*, 12 2013.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *CoRR*, vol. abs/1406.4729, 2014.

[11] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015.

[12] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[13] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016.

[14] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," *CoRR*, vol. abs/1801.05134, 2018.

[15] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018.

[16] D. Hoiem, Y. Chodpathumwan, and Q. Dai, "Diagnosing error in object detectors," in *Computer Vision – ECCV 2012* (A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, eds.), (Berlin, Heidelberg), pp. 340–353, Springer Berlin Heidelberg, 2012.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[19] S. Vinodababu, "Ssd: Single shot multibox detector | a pytorch tutorial to object detection."

[20] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013.

[21] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016.

[22] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016.

[23] W. Liu, A. Rabinovich, and A. C. Berg, "ParseNet: Looking Wider to See Better," *arXiv e-prints*, p. arXiv:1506.04579, Jun 2015.

[24] CyberAILab, "A closer look at yolov3."

[25] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[27] R. Rothe, M. Guillaumin, and L. Van Gool, "Non-maximum suppression for object detection by passing messages between windows," vol. 9003, 04 2015.

[28] "Best of both worlds: Human-machine collaboration for object annotation," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, (United States), pp. 2121–2131, IEEE Computer Society, 10 2015.

[29] I. of Computer Science, T. of Peking University, and I. of Digital Publishing of Founder RD Center (China), "Marmot dataset." `http://www.icst.pku.edu.cn/cpdp/sjzy/index.htm`.

[30] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[31] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *AISTATS*, 2011.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[33] J. Redmon, "Darknet: Open source neural networks in c." `http://pjreddie.com/darknet/`, 2013–2016.

[34] S. Carrión, "Detection of mathematical expressions in scientific papers." https://github.com/salvacarrion/yolo4math.

[35] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[36] D. Mahajan, R. B. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," *CoRR*, vol. abs/1805.00932, 2018.

[37] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019.

[38] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," *CoRR*, vol. abs/1709.01507, 2017.

[39] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," 2019.

[40] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *CoRR*, vol. abs/1708.02002, 2017.