IMPLEMENTATION OF FASTER R-CNN APPLIED TO THE DATASETS COCO AND PASCAL VOC

Pedro de Carvalho Cayres Pinto

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: José Gabriel Rodríguez Carneiro Gomes

Rio de Janeiro
Março de 2019

IMPLEMENTATION OF FASTER R-CNN APPLIED TO THE DATASETS COCO AND PASCAL VOC

Pedro de Carvalho Cayres Pinto

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

____
Prof. José Gabriel Rodríguez Carneiro Gomes, Ph.D.

____
Prof. Mariane Rembold Petraglia, Ph.D.

____
Eng. Leonardo de Oliveira Nunes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2019

Pinto, Pedro de Carvalho Cayres

Implementation of Faster R-CNN Applied to the Datasets COCO and PASCAL VOC/Pedro de Carvalho Cayres Pinto. – Rio de Janeiro: UFRJ/COPPE, 2019.

XV, 69 p.: il.; 29,7cm.

Orientador: José Gabriel Rodríguez Carneiro Gomes

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2019.

Referências Bibliográficas: p. 63 – 69.

1. object detection. 2. convolutional neural network. 3. computer vision. 4. deep learning. I. Gomes, José Gabriel Rodríguez Carneiro. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Dedico este trabalho à minha família.

Agradecimentos

Agradeço à minha família pelo apoio e pelo carinho.

A todos os meus amigos e professores.

Ao meu orientador, José Gabriel, pelos ensinamentos e conselhos que muito me auxiliaram no desenvolvimento deste trabalho.

Aos colegas Leonardo Mazza, Olavo Sampaio, Igor Quintanilha, Roberto Estevão, à professora Mariane Petraglia e ao professor José Gabriel Gomes pelas reuniões, sempre muito produtivas. Também agradeço aos colegas do PADS, em especial à Fernanda Duarte e ao Gustavo Nunes.

Ao Luiz Rennó, que percorreu grande parte desta caminhada comigo.

Também agradeço ao Laboratório de Processamento Analógico e Digital de Sinais (PADS) por fornecer a infraestrutura necessária à elaboração deste trabalho.

Por fim, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio financeiro.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO DE FASTER R-CNN E APLICAÇÃO ÀS BASES DE DADOS COCO E PASCAL VOC

Pedro de Carvalho Cayres Pinto

Março/2019

Orientador: José Gabriel Rodríguez Carneiro Gomes

Programa: Engenharia Elétrica

Esta dissertação apresenta implementações de dois detectores de objetos baseados em redes neurais convolucionais: Faster R-CNN e Faster R-CNN com FPN. É feita uma breve introdução ao aprendizado de máquinas, em seguida há uma explicação sobre a tarefa de classificação de imagens, onde são apresentadas as arquiteturas VGG-16 e ResNet-101, assim como uma explicação detalhada sobre a tarefa de detecção de objetos e sobre os métodos que levaram ao desenvolvimento do Faster R-CNN. Após isso, há uma discussão sobre as implementações como um todo, apresentando todos os parâmetros utilizados, assim como a infraestrutura utilizada para construir as redes e as diferenças com relação às implementações originais. Então, são realizados três experimentos, utilizando as bases de dados COCO e PASCAL VOC para treino e teste, e os resultados são comparados com os dos trabalhos originais com a métrica da média das precisões médias (mAP), e estes resultados são analisados. Também são feitas algumas considerações sobre o tempo de inferência dos métodos. Finalmente, alguns exemplos de detecção da melhor rede são apresentados. No experimento feito na base COCO, o detector FPN obteve um mAP@[.5, .95] de 38.1% e mAP@0.5 de 61.1% no conjunto COCO test-dev (um modelo mais recente, RetinaNet com ResNeXt-101-FPN, obtém 40.8% de mAP@[.5, .95] e 61.1% de mAP@0.5 no conjunto COCO test-dev). O código está disponível em: `https://gitlab.com/pedrocayres/faster_rcnn_pytorch`.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPLEMENTATION OF FASTER R-CNN APPLIED TO THE DATASETS COCO AND PASCAL VOC

Pedro de Carvalho Cayres Pinto

March/2019

Advisor: José Gabriel Rodríguez Carneiro Gomes

Department: Electrical Engineering

This dissertation presents implementations of two object detection systems, Faster R-CNN and Faster R-CNN with FPN, based on convolutional neural networks. There is a brief introduction to machine learning, followed by an explanation of the image classification task, where the VGG-16 and ResNet-101 architectures are presented, as well as detailed explanations of the object detection task and the methods that led to the development of Faster R-CNN. Next, the implementation of the algorithms is discussed thoroughly, specifying the parameters and the framework used to build the networks, and mentioning differences with the original. Then, three experiments are performed, using the COCO and PASCAL VOC datasets for training and testing, and the results, on the mean average precision (mAP) metric, are compared with the original counterparts of the methods. The obtained results are discussed and some considerations are made about the inference time of the implementations. Finally, detection examples of the most accurate implementation are presented. The FPN detector achieved 38.1% mAP@[.5, .95] and 61.1% mAP@0.5 on the COCO test-dev set (a more recent model, RetinaNet with ResNeXt-101-FPN, achieves 40.8% mAP@[.5, .95] and 61.1% mAP@0.5 on the COCO test-dev set). The code is available at: `https://gitlab.com/pedrocayres/faster_rcnn_pytorch`.

Contents

List of Figures **x**

List of Tables **xiv**

1 Introduction **1**

1.1 Description . . . 2

2 Theory **3**

2.1 Artificial Neural Networks . . . 3

2.1.1 Optimization . . . 5

2.1.2 Datasets . . . 7

2.1.3 Graph . . . 7

2.1.4 Convolutional Neural Networks . . . 8

2.1.5 Regularization . . . 11

2.1.6 Classification vs regression . . . 12

2.2 Image Classification . . . 13

2.2.1 VGG . . . 14

2.2.2 ResNet . . . 15

2.2.3 Architecture . . . 16

2.3 Object Detection . . . 17

2.3.1 Intersection over Union . . . 19

2.3.2 Mean Average Precision . . . 19

2.3.3 R-CNN . . . 20

2.3.4 Fast R-CNN . . . 23

2.3.5 Faster R-CNN . . . 26

2.3.6 Feature Pyramid Networks . . . 30

2.3.7 Mask R-CNN . . . 34

3 Implementation **37**

3.1 Framework . . . 37

3.2 Non-maximum suppression . . . 38

3.3 Faster R-CNN . . . 39

3.3.1 Backbone 39
3.3.2 Head 40
3.3.3 RPN 40
3.3.4 Classification and regression 42
3.3.5 Postprocessing 42
3.4 FPN 42
3.4.1 Backbone and head 43
3.4.2 RPN 43
3.4.3 Classification and regression 44
3.5 Hyperparameters 44

4 Results **46**
4.1 Datasets 46
4.1.1 COCO 46
4.1.2 PASCAL VOC 47
4.2 Preprocessing 47
4.3 Hyperparameters for training 47
4.4 Experiments on COCO 48
4.4.1 Optimization 48
4.4.2 Results 48
4.5 Experiments on PASCAL VOC 49
4.5.1 Optimization 49
4.5.2 Results 50
4.6 Time profiling 50
4.7 Discussion 51
4.8 Examples 52
4.9 Improvements 54
4.9.1 Pre-NMS filters 54
4.9.2 Mini-batch sampling 54
4.9.3 Loss function 57
4.9.4 Inference 57
4.9.5 Optimization 58
4.9.6 Experiments on COCO 58
4.9.7 Experiments on PASCAL VOC 59

5 Conclusion **61**
5.1 Future Work 61

Bibliography **63**

List of Figures

2.1 Representation of an ANN. The inputs are in blue, green circles represent hidden artificial neurons with activation functions, and red circles symbolize the output neurons, that can have activation functions or not, depending on the problem. 4

2.2 Convolution of an input tensor with two filters. The filters and their respective feature maps are represented in different colors. The arrows represent the dot product of the filter / sliding window and the subwindow of the input at that position. 10

2.3 Example of a 2×2 max pooling with stride 2 performed in a 4×4 feature map. The max operation is performed in each colored window. This illustrates only one slice of the input and output tensors. The operation is applied independently for each channel. 10

2.4 An example of image classification. When classified with ResNet-101 (pre-trained on ImageNet), the top-5 labels for this image are "tiger_cat, Egyptian_cat, tabby, lynx, Siamese_cat". 13

2.5 Building blocks of ResNet. The building block in the left is the one used in ResNet-18/34. The bottleneck building block in the right is used for ResNet-50/101/152. If the input has $4F$ channels and the same spatial dimension as the output the shortcut connection is the identity, otherwise the shortcut connection is a 1×1 convolutional layer that projects the input to the correct dimensions. 16

2.6 An example of object detection with FPN on an image from the COCO validation set. The top image shows all ground-truth annotations. The bottom image shows the detections made by FPN with their respective confidences. Detections were filtered with a threshold of 0.5. 18

2.7 Example of a precision / recall curve using the detection vector (ordered by score): [T, T, T, F, T, T, F, T, F, F], where T is true positive and F is false positive, with a total of 6 ground-truths. 21

2.8 Illustration of the R-CNN architecture. (1) loads an image; (2) around 2000 proposals are generated by selective search and warped to a fixed size, to be processed by a CNN; (3) features are extracted using a CNN; (4) the system classifies the features with binary SVMs (one per class) 21
2.9 Illustration of the Fast R-CNN architecture. Unlike R-CNN, the features of the convolutional layers are extracted once, then the regions proposed by selective search are projected to the feature maps of the last layer. These projected regions are then passed to the region-specific fully connected layers, for classification and regression, by the RoI pooling layer. The last module is applied once per region of interest, but the process is made much lighter by removing the need to call the full CNN on each proposal. 24
2.10 A region of interest of dimensions 7×8 in the feature maps is extracted with RoIPool to 3×3 feature maps. Each colored window represents one value in the output. The output is determined by taking the maximum inside each window, for each channel independently. 25
2.11 Illustration of the Faster R-CNN architecture. In Faster R-CNN the proposal generation module (RPN) shares layers with a Fast R-CNN system. 27
2.12 Overview of the RPN specific layers. A sliding window scans the feature maps to generate features related to specific positions at the image. These features are scored with a fully connected layer to define how likely it is that a bounding box of specific size (an anchor) placed at that position fit well an object, for each of the k pre-defined anchors. A sibling layer for regression is also used for each of the anchors. These extra layers are implemented with a fully convolutional network using a 3×3 convolution followed by ReLU and two sibling 1×1 convolutions for classification and regression. 28
2.13 Different prediction models for object detection: (a) Prediction is done from a feature pyramid built upon an image pyramid. (b) Classic CNN, prediction is calculated from the feature maps from only one convolutional layer. (c) Feature pyramid prediction, but the feature pyramid is built using the feature hierarchy directly. (d) An improvement of (c) by including top-down and lateral connections to get features with strong context in every level. It is also much less computationally expensive than (a), having inference speed closer to (b) and (c) while having better performance. 31

2.14 Layers introduced by the FPN. The lateral connection is a 1×1 convolutional layer that equalizes the number of channels at all pyramid levels, while the top-down connection is a $2\times$ upsample that introduces deeper semantic information into the lower levels. 32
2.15 This figures illustrates the RoIAlign process for one region of interest. The dashed grid represents the feature maps, the solid grid represents a projected region of interest, where the rectangles are the bins, and the blue dots are the samples that are computed with bilinear interpolation. 35

4.1 Random samples of COCO minival detected by the trained FPN (with a threshold of 0.5). In the top-left image, some laptops were correctly identified, while some were confused with chairs, and one of the detected laptops has a misplaced bounding box. There were some keyboard detections, but laptop keyboards are not considered keyboards in the ground-truth. The backpack was mistook for a handbag / suitcase. The tangled wires were detected as a bicycle and part of the router was mistaken for a book. In the top-right image, the persons and the racket were detected correctly, the detector even found people in the stands. In the bottom-left image, one person and the motorcycle were identified correctly, and the handbag was incorrectly classified. In the bottom-right image, all elephants were identified correctly. 53
4.2 Detections made by the FPN on the frames of a video of a street crossing. Only detections of persons, bicycles, cars, motorcycles, buses, trucks, traffic lights and stop signs, that have a confidence greater than 0.5, are displayed. The detector consistently detects a bicycle in the middle pole, and a person in the top-right of the images near the safety net. In the top-left image, it detected some cars and people correctly, but it misclassified a person and a street sign as traffic lights. In the top-right image, the detector identified all cars correctly, albeit with one false positive. It also detected the traffic light in the middle of the image. In the bottom-left image, the truck was classified as a car and generated three false detections. Moreover, a person in the sidewalk was also detected. In the bottom-right image, almost all cars were detected. It also detected the traffic light and people riding bicycles, but missed the bicycles themselves. 55

4.3 Detections made by the FPN on the frames of a video of a street crossing at night. In all images the detector finds the traffic light, but incorrectly detects persons in the right and left side of the image. In the top-left image, four of the five cars are detected, although regression seems worse than in the daytime video. The top-right image presents two incorrect car detections. One of them is a false positive generated by a motorcycle. In the bottom-left image, there is one more incorrect person detection (at the car headlight), but the detector correctly identified the pedestrian traffic light in the top. In the bottom-right image, four out of six cars are detected. 56

List of Tables

2.1 Configuration of the layers of VGG-16. All convolutions are followed by ReLU and have zero-padding 1, such that the input and output dimensions are the same at each stage. The hidden fully connected layers also have ReLU activation. Input dimensions have the channels omitted. 15

2.2 Configuration of the layers of ResNet-101. All convolutions are followed by ReLU and have padding $\lfloor \frac{F}{2} \rfloor$ where F is the filter size, such that the output is the same size as the input when stride is 1. Stage k is denoted as conv$k$ (or Ck) and the first layer of each stage has stride 2. There is a shortcut connection between the input and output of every convolutional block. 17

3.1 Summary of hyperparameters used in the implementations of Faster R-CNN and FPN. Some hyperparameters are fixed across all implementations: 512 RPN channels, 3 RPN kernel size, 0.7 RPN NMS threshold, $\{(1:1),(1:2),(2:1)\}$ anchor ratios. 45

4.1 Detection results on the COCO validation set. Training was done on the COCO training set. The size column is the size of the smallest side of the image input of that model, while the RoIOp. column determines the donwsampling operation performed on the regions of interest of that model. 49

4.2 Detection results on the PASCAL VOC 2007 test set. Training was done on the PASCAL VOC 2007 trainval set. 50

4.3 Detection results on the PASCAL VOC 2007 test set. Training was done on the union of PASCAL VOC 2007 trainval and PASCAL VOC 2012 trainval sets. 50

4.4 Time spent in each module of the networks for one detection during test time. RPN time refer to the RPN-specific layers and calculating the proposals, and NMS time is the time spent on NMS in the proposals from RPN only. Other includes RoIAlign, softmax and class specif NMS. These values are average values computed over 1000 samples. All tests were done on a Nvidia GTX 1080 Ti GPU and an Intel Core i7-6850K CPU. 51
4.5 Detection results on the COCO validation set with the updated methods. Training was done on the COCO training set. 59
4.6 Detection results on the COCO test-dev set with the updated methods. 59
4.7 Detection results on the PASCAL VOC 2007 test set with the updated methods. Training was done on the PASCAL VOC 2007 trainval set. 60
4.8 Detection results on the PASCAL VOC 2007 test set with the updated methods. Training was done on the union of PASCAL VOC 2007 trainval and PASCAL VOC 2012 trainval sets. 60

Chapter 1

Introduction

This work presents the implementation of image-based object detectors. An image-based object detector is an algorithm which, given an image, is capable of locating objects of a set of categories, called classes, and finding a bounding box that determines its position in the image. Object detection can be used in, for example, the development of autonomous vehicles [1], dense captioning systems [2], instance segmentation systems [3], photography applications, surveillance systems, military applications and traffic control.

These systems can be built upon a variety of techniques such as CNN (convolutional neural network) [4–14], SIFT (scale-invariant feature transform) [15, 16], HOG (histogram of oriented gradients) [17, 18], Haar features [19], SVM (support vector machine) [20], among others.

The objective of this work is to implement object detection algorithms based on CNNs, that have recently been shown to produce state-of-the-art performance in tasks such as image classification [21, 22], object detection [4–14], semantic segmentation [23], and instance segmentation [3]. This works also aims to provide a more detailed discussion on the implementation of these algorithms, covering details not presented in the original papers.

The implementation is mainly based on the articles about Faster R-CNN [7] and FPN (feature pyramid network) [8], where the CNN is used for both proposal generation and classification. The preceding works about R-CNN (regions with CNN features) [4] and Fast R-CNN [6] are mentioned, as well as the more recent Mask R-CNN [3], that uses a similar framework for instance segmentation.

These methods can be decomposed into three modules: feature extraction using multiple convolutional layers; proposal generation, that returns regions where, possibly, objects are present; and classification of each of those proposals in a set of pre-defined categories (including background, for filtering bad proposals), as well as a regression of the coordinates of the bounding box of each object.

Other systems with good performance, which are also based on CNNs, include

SSD (single shot detector) [9], DSSD (deconvolutional SSD) [10], YOLO (you only look once) [11, 12], R-FCN (region-based fully convolutional network) [13] and RetinaNet [14].

Object detection systems based on CNNs require large datasets to achieve high precision so, in this work, the parameters of the implemented models are trained and tested using the COCO (common objects in context) [24] and the PASCAL (pattern analysis, statistical modelling and computational learning) VOC (visual object classes) [25, 26] object detection datasets.

1.1 Description

Chapter 2 discusses the basic theory on artificial neural networks, artificial neurons, linear and convolutional layers, optimization, regularization, datasets, and the classification and regression problems. Moreover, the problem of image classification is introduced, including details about the VGG and ResNet CNN that are used in the object detectors implemented. Finally, the object detection problem is discussed and the R-CNN, Fast R-CNN, Faster R-CNN and FPN systems are detailed. Mask R-CNN is briefly introduced as well.

Chapter 3 treats in detail the implementation of the Faster R-CNN and FPN detectors, specifying the hyperparameters used in the experiments. Aspects of the detectors are revisited with more focus given to the implementation.

Chapter 4 describes the PASCAL VOC and COCO datasets, as well as the experiments performed in this work. The results are presented and compared with the ones from the published papers. This chapter also provides examples of detections made by the trained networks in the test dataset and in video frames.

In Chapter 5, some final considerations are made with respect to the results and future work proposals are given to improve the implementation accuracy and inference time, and to increment to the work with new modules.

Chapter 2

Theory

This chapter encompasses the theory necessary for the development of object detection with convolutional neural networks that are implemented in this work. It starts with a discussion about the basics of the theory on artificial neural networks, where it briefly describes artificial neurons, multilayer perceptrons, activation functions, convolutional layers, optimization, regularization, classification and regression. Then, it goes on to a description of the image classification problems, presenting the VGG and ResNet CNN architectures that will be used later on for object detection. Finally, the problem of object detection is presented, the intersection over union and mean average precision metrics are defined, and the R-CNN, Fast R-CNN, Faster R-CNN and FPN object detection systems are explained in detail. A brief introduction to Mask R-CNN and the instance segmentation problem is also given.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) [27] are systems designed to be able to learn how to solve a given task. This task can be described as returning the correct data outputs when receiving given data inputs. Examples of tasks that can be solved with ANN are image classification, object detection, image captioning [28], instance segmentation, speech recognition [29], and machine translation [30].

Tasks can usually be interpreted as finding a function $\tilde{f}$ that approximates the function $f \colon D \subseteq \mathbb{R}^M \to I \subseteq \mathbb{R}^N$ that corresponds to the ideal mapping of the task from a set D to a set I. For example, in the case of image classification, elements of D could be the values of the pixels of an image, while the output in I could be the probabilities of the image belonging to a certain category. In this case, the image of a person would be mapped by f to a value of 1 in the person category and to 0 in all other categories. Note that the domain D can be very hard to define, so the ANNs seen here will have as domain simply $\mathbb{R}^M$, but the domain D is implicitly used in the comparisons because dataset entries are by definition sampled from it.

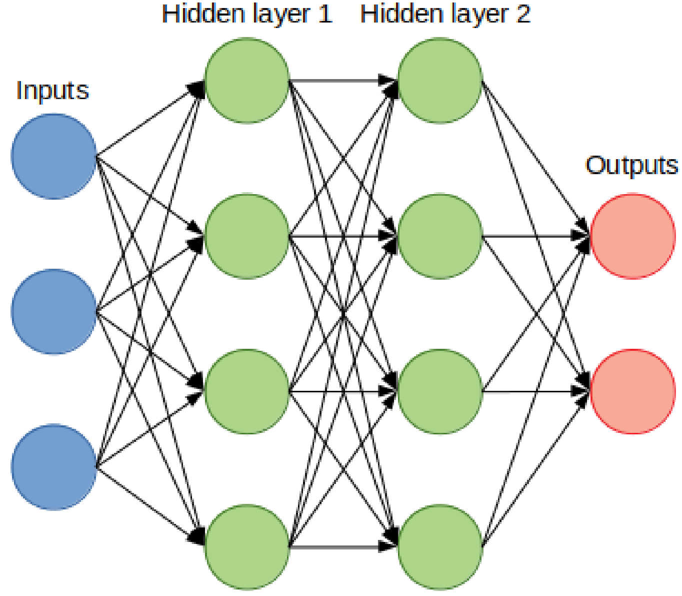


Figure 2.1: Representation of an ANN. The inputs are in blue, green circles represent hidden artificial neurons with activation functions, and red circles symbolize the output neurons, that can have activation functions or not, depending on the problem.

In this context, ANN is an approach that tries to solve this problem with a function $\tilde{f}$ defined as in Equation (2.1). Here $\tilde{f}$ is a parameterized function and the task simplifies to finding a value of the parameters θ that approximates f.

$$\begin{aligned} \tilde{f} \colon \mathbb{R}^M \times \mathbb{R}^P &\to I \subseteq \mathbb{R}^N \\ x, \theta &\mapsto y \end{aligned} \tag{2.1}$$

ANNs are composed by artificial neurons that apply a simple function to their data input. The generation of an output by an artificial neuron is called an activation. Artificial neurons can pass their output to subsequent ones, forming a directed graph structure, mapping the input to the output of the ANN. This can be seen in Figure 2.1.

Equation (2.2) shows an example of an artificial neuron that applies the scalar product to a vector input $\mathbf{x}$. Here, the vector $\mathbf{w}$ is called a weight vector and b the bias of the artificial neuron. These parameters define the operation the neuron performs and may be changed, in order to improve the results of the ANN, through optimization.

$$n(\mathbf{x}, \mathbf{w}) = \sum x_i w_i + b \tag{2.2}$$

Often the input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is modified to $\mathbf{x} = (x_1, x_2, \ldots, x_n, 1)$. This simplifies Equation (2.2) into Equation (2.3).

$$n(\mathbf{x}, \mathbf{w}) = \sum x_i w_i \tag{2.3}$$

It is common to group artificial neurons that perform similar operations into groups called layers. This way, the output of a layer with neurons that apply the function defined in Equation (2.3) is simply the multiplication of the input by a weight matrix. This layer is called the linear layer and applies the function $g \colon \mathbb{R}^M \times \mathbb{R}^{M \times N} \to \mathbb{R}^N$ defined in Equation (2.4), where $\mathbf{x} \in \mathbb{R}^M$ is the input, $\mathbf{W} \in \mathbb{R}^{M \times N}$ is the matrix of weights, that is, the parameters of this layer to be optimized.

$$g(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x} \tag{2.4}$$

Since compositions of linear functions are linear, to access a larger space of functions non-linearities are introduced. Activation functions are functions applied to the outputs of some layers, with the purpose of introducing non-linearities into the model. There are many types of non-linearities. Initially ANNs used hyperbolic tangent, sigmoid as activation functions, but such functions introduce limitations in the optimization of the ANN because they can lead to optimization issues [31]. The ReLU [27, 32] activation function (Equation (2.5)) is more robust to these problems and is the one used in this work.

$$\mathrm{ReLU}(\mathbf{x}) = \max(\mathbf{x}, \mathbf{0}) \tag{2.5}$$

2.1.1 Optimization

A fixed ANN topology comprises a set of ANNs – functions from the input space to the output space – parameterized by weights. To train an ANN for a given task is to search (i.e. to choose weights for) a function in that topology, so that the function performs well in returning the desired outputs as specified in the task. In general, no member of the topology returns the correct output for every input, and thus the ANN are compared in terms of a loss function. Therefore, to solve a problem with ANN, it is usual to first implement an ANN topology, and then train the ANN by finding the best weights.

Training can be carried out in different ways, depending on the problem. When a dataset of labeled input-output pairs is provided, it is possible to perform *supervised training* [27]. In supervised training a loss function is defined to measure dissimilarity between the labeled outputs and the outputs computed by the ANN. In this case, training is performed by minimizing the average loss over the entries in the training dataset, by changing the weights of the ANN using some optimization algorithm. Training an ANN without labeled outputs is also possible, in some cases,

and is called *unsupervised training* [27]. When only some of the outputs are labeled another strategy that can be implemented is semi-supervised training. There is also *reinforcement learning* [27], in which the ANN describes how an agent takes action in an environment and the training is driven by trying to maximize the rewards the environment gives to the agent. In this work all networks are trained with supervised training and are thus optimized using datasets containing annotated labels.

In supervised training the objective function J can be defined as in Equation (2.6), where l is the per sample loss function of the ANN, f is the ANN function, $\mathbf{x}$ is the input and $\mathbf{y}$ is the true output. To train the ANN is to search for parameters $\boldsymbol{\theta}$ that minimize this function.

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}} l(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}) \tag{2.6}$$

However, since the information regarding the input-output pairs is known only for given samples in a dataset, the function J can only be computed by the approximation given in Equation (2.7).

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=0}^{N} l(f(\mathbf{x_i}, \boldsymbol{\theta}), \mathbf{y_i}) \tag{2.7}$$

Since functions that compose an ANN are differentiable, except at a finite number of points, the gradient of the loss with respect to the weights can be calculated using the chain rule. This way, gradient-based optimization algorithms such as gradient descent can be used to train the ANN by optimizing Equation (2.7). In practice optimizing this objective directly is expensive for large datasets, so it is more common to optimize with the *stochastic gradient descent* (SGD) algorithm [27], that trains the network with just a few samples called *mini-batches*, that are used to estimate the objective function. One update step of SGD is described in Equation (2.8), where t is the iteration step, $\hat{J}$ is the estimate of J in the mini-batch, and B is the size of the mini-batch, and η is the learning rate. The learning rate is an important optimization hyperparameter, since large values can make the objective function training series diverge and small values make convergence slow and susceptible to local minima.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \hat{J}(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} \sum_{i=0}^{B} l(f(\mathbf{x_i}, \boldsymbol{\theta}), \mathbf{y_i}) \tag{2.8}$$

The SGD used in this work uses momentum, denoted as μ, whose update rule is defined in Equation (2.9) [33][1]. Note that the variable vector $\mathbf{v}$ accumulates steps from previous iterations, with the idea of accelerating convergence on low curvature

[1]The implementation found in PyTorch [34] is slightly different, but follows the same idea.

regions of the objective function, where the gradients are small and do not change directions much, while if the curvature is high the gradients' directions will vary more often and the gradient term will be larger.

$$\begin{aligned} \mathbf{v}_{t+1} &= \mu \mathbf{v}_t - \eta \nabla_{\boldsymbol{\theta}} \hat{J}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1} \end{aligned} \tag{2.9}$$

2.1.2 Datasets

To train supervised tasks, datasets are necessary to provide samples with annotated inputs and their corresponding outputs. Datasets are usually divided in three types: train, validation and test.

The train set is usually the largest dataset and is used for optimizing the parameters of the model. The performance of the system in this set usually improves as more updates are performed in the parameters. It is important to note that the performance of the model, on the designed task, evaluated on the training set is not a good measure for estimating the performance the model will have on samples from outside this dataset, since the model may be biased. On this note, a better way of evaluating the model is the validation set.

A validation set is a dataset that is not used for updating the parameters of the model, but that can be used for tuning the hyperparameters. This set is useful for evaluating model generalization, that is, model performance on yet unknown samples that it may be used to infer from in the future. Hyperparameter tuning is the task of modifying the hyperparameters of the model so that it achieves better generalization. This is done by calculating the cost function in the validation set for multiple models with different parameters and choosing the best one.

The test set is not used at any moment during development and hyperparameter tuning, so as that the model remain unbiased on this dataset. Published results usually report performance on this dataset.

2.1.3 Graph

An ANN can be interpreted as a graph [27], which is important because some frameworks, such as PyTorch [34] (the framework used in this work) and TensorFlow [35], use this property to efficiently compute the gradients within an ANN. In this interpretation, the vertices and edges are the ANN inputs, weights and functions (for example, artificial neurons, layers and loss functions), which have as inputs the outputs of the edges that are directed at them.

The ANNs discussed in this work are all *feedforward*, meaning that their graphs representations are acyclic directed graphs.

To compute the outputs of a *feedforward* ANN is to run an algorithm that walks through the graph calculating the output of every vertex. The algorithm computes every vertex accessible through the already computed ones, applying the functions described on them. At first the input and weights are given and after the algorithm finishes the output and all hidden variables are computed.

The gradients are computed in the backward direction using a similar algorithm that computes the gradients with respect to the weights based on the chain rule. Starting from the loss function vertex, with gradient 1, a vertex, holding the variable y, sends to an input vertex, holding the variable x, the gradient $\frac{dL}{dx} = \frac{dL}{dy}\frac{dy}{dx}$, iterating this process until the gradients on all weights are computed.

2.1.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) [27] employ convolutional layers, which have fewer parameters and computational cost than linear layers, to take advantage of the spatial distribution of pixels in images, achieving state-of-the-art results in tasks such as image classification, detection and segmentation.

The convolutional layer applies a group of filters to its inputs. In the single dimensional case a convolution filter, with weights $\mathbf{w} \in \mathbb{R}^P$ and bias $b \in \mathbb{R}$, is defined as g in Equation (2.10). Note that this equation can be written in matricial form such as in Equation (2.11) (where x is multiplied by a matrix $\mathbf{W} \in \mathbb{R}^{N-P+1 \times N}$ whose values are defined as $W_{i,j} = w_{j-i}$, when $(j - i) \in \{0, 1, ...P - 1\}$, otherwise $W_{i,j} = 0$), hence it can be seen as a modification of the linear layer, implementing weight sharing between the neurons.

$$g_i(x) = \sum_{j=0}^{P-1} x_{i+j} w_j + b \tag{2.10}$$

$$g(x) = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_{P-1} & 0 & 0 & \cdots & 0 \\ 0 & w_0 & w_1 & \cdots & w_{P-2} & w_{P-1} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & w_{P-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix} \tag{2.11}$$

Convolutional layers are particularly useful when handling image tasks, where the input is an image that can be converted to a float tensor with dimensions $H \times W \times 3$, with H and W being, respectively, the height and width of the image, measured in pixels, and each value in the tensor is the saturation of some color in

RGB, represented along the third dimension. In this case the convolutional filter g assumes the form of Equation (2.12), receiving two-dimensional inputs with multiple channels. In general the input has multiple spatial dimensions and one channel dimension and the convolutional filter is a sliding window that applies a dot product of its weights and adds a bias to the sub-windows of the input tensor it passes through (thus, the number of channels of the weight tensor is equal to the number of channels in the input tensor).

$$g_{i,j}(x) = \sum_{c=0}^{C-1} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} x_{i+k,j+l,c} w_{k,l,c} + b \tag{2.12}$$

The output of a convolutional filter is called a feature map. Convolutional layers stack the outputs of multiple filters, so, for example, an input $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ generates an output $\mathbf{y} \in \mathbb{R}^{H_l \times W_l \times C_l}$, where C_l is the number of filters and feature maps generated.

The operation defined in (2.12) is called a convolutional filter with stride 1. This is because this convolution can be visualized as a window sliding (the tensor w) over the tensor of input feature maps, calculating the dot product at each position, generating one output value for every step, as can be seen in Figure 2.2. Other stride values S can be used and, in that case, an output $g_{i,j}$ is generated when the window slides iS positions right and jS positions down in the tensor (starting from the top-left position in the input), which is used for downsampling.

Another hyperparameter in the convolutional layer is padding. Since the function in Equation (2.12) is only defined for values of (i, j) where the sum is valid, the spatial dimensions of $g(x)$ are smaller than that of x whenever the window size of the convolution is greater than 1. To prevent this, the spatial dimensions of input x are amplified with a border of new values (usually zeros). This process is known as padding and a layer that implements a padding of P appends P columns on the left and right and P rows on the top and bottom of the input x. The output spatial dimension M of a convolutional layer can then be calculated as a function of the input size N, window size F, stride S and padding P using Equation (2.13). CNNs commonly employ layers with the same input and output dimensions, for example, by using $S = 1$, $F = 3$ and $P = 1$.

$$M = \frac{N + 2P - F}{S} + 1 \tag{2.13}$$

Pooling

The pooling layer is another operation commonly employed in CNNs [27]. In pooling, a sliding window is passed over the input feature maps, computing one value for each window position and feature map. This operation has no parameters and the

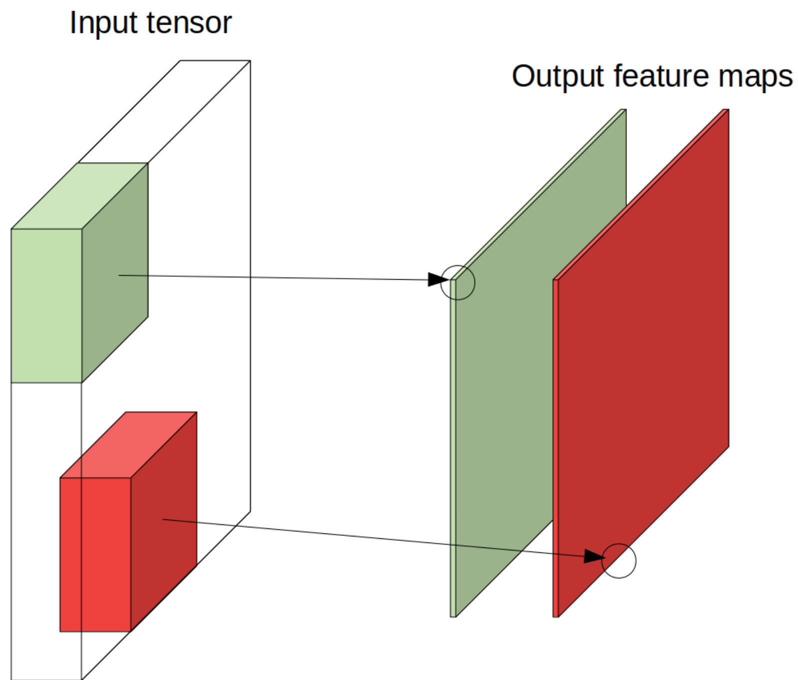


Figure 2.2: Convolution of an input tensor with two filters. The filters and their respective feature maps are represented in different colors. The arrows represent the dot product of the filter / sliding window and the sub-window of the input at that position.

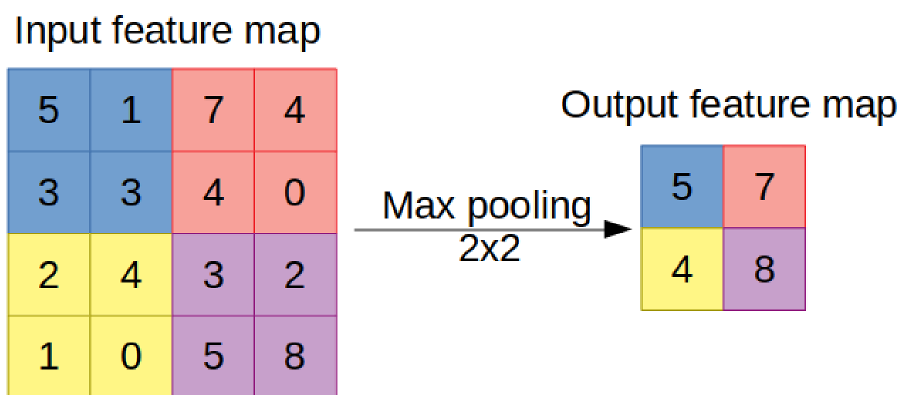


Figure 2.3: Example of a 2×2 max pooling with stride 2 performed in a 4×4 feature map. The max operation is performed in each colored window. This illustrates only one slice of the input and output tensors. The operation is applied independently for each channel.

value computed by the sliding window is usually the maximum value or the average value, called max pooling and average pooling, respectively. Note this operation is calculated independently for each feature map in the window, so the number of channels is preserved. Like the convolutional layer it also has the hyperparameters of window size and stride. Max pooling with window size 2×2 and stride 2 is quite usual. An example of max pooling can be seen in Figure 2.3.

The purpose of the pooling layer is to reduce the number of features in deeper layers of the network and also to introduce translation invariance.

A common setup for a CNN involves multiple convolutional layer blocks with ReLU activation, with periodical pooling layers in between for downsampling, as will be seen in VGG and ResNet.

For max pooling, the gradient flows through the maximum position in each sub-window in the backward pass, and all other positions have zero gradient. That can be seen as the max pooling layer acting as a switch in the backward direction for each sub-window. For average pooling, the feature map sub-windows are averaged, thus, in the backward direction, the gradient is equally split among all positions in each sub-window.

2.1.5 Regularization

It is desired that the model obtained from training the ANN is generalized, i.e., that it performs well on data that is not in the training dataset. When the weights become specialized in solving the training dataset samples it is said that *overfitting* occurred. When overfitting occurs, the average loss of the validation dataset starts increasing after training a large number of iterations, while the training dataset loss keeps decreasing. The act of attenuating this behaviour is known as regularization. Examples of regularization techniques include L_2 regularization [27], dropout [36] and batch normalization [37].

To perform L_2 regularization in an ANN is to add a small multiple λ of the square of its weights to the loss function, such as in Equation (2.14), where l_r is the regularized loss updated from l. In this way the optimizer tries to minimize the L_2 norm of the parameters, thus reducing weight variance. This regularization is also called weight decay because a term of the form $(1 - \eta\lambda)\boldsymbol{\theta}$ appears in the SGD update step. The formula for the gradient is given in Equation (2.15).

$$l_r(\mathbf{y}, \mathbf{t}) = l(\mathbf{y}, \mathbf{t}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2 \tag{2.14}$$

$$\nabla_{\boldsymbol{\theta}} l_r(\mathbf{y}, \mathbf{t}) = \nabla_{\boldsymbol{\theta}} l(\mathbf{y}, \mathbf{t}) + \lambda\boldsymbol{\theta} \tag{2.15}$$

Another idea for regularization is dropout, where some fraction of the parameters are disabled each training step, so as to enable the network to have some redundancy and prevent it from learning weights too specialized for solving the training set.

Batch normalization prevents the distributions of input variables inside the layer from varying too much, by normalizing these variables using mean and variance estimates based on the mini-batch being processed. A layer that allows the network to undo this normalization is also included, to let the network learn the range of values the features can assume. Because of the estimates, this process is noisy and this also helps with generalization.

2.1.6 Classification vs regression

ANNs are usually used to solve tasks of classification or regression. The classification task involves finding which class, from a predefined set of categories, a given input belongs to, and is thus a discrete output problem. On the other hand, the regression task is about finding a function that approximates well another function, whose values are unknown except for some given input / output pairs.

When the objective is to classify an input in only one category, a common modelling scheme consists in using a vector output whose values add up to 1. This can be interpreted as the probability of the input belonging to that class, and the class identified by the ANN is the one with the highest probability. These outputs are generated by passing scores generated by the ANN through a *softmax* layer [27], that normalizes the scores $\mathbf{y}$ as in Equation (2.16).

$$\text{Softmax}(\mathbf{y})_i = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{2.16}$$

Since the output is considered a probability distribution, the ANN parameters $\boldsymbol{\theta}$ can be optimized with the maximum likelihood estimator method given in Equation (2.17), where $\mathbf{x}$ is the ANN input and c is the correct class.

$$\hat{\boldsymbol{\theta}} \in \arg\max_{\boldsymbol{\theta}} P_{\boldsymbol{\theta}}(C = c|\mathbf{x}) \tag{2.17}$$

Working with the log likelihood is usually more convenient, because it simplifies the problem to minimizing the *cross entropy loss* [27] function l in Equation (2.18), where $\mathbf{p}$ is the vector of probabilities given by the ANN and c is the true class.

$$l(\mathbf{p}, c) = -\log p_c \tag{2.18}$$

Regression involves finding real-valued functions and the formulation is more straightforward, using the outputs of linear layers directly. The loss function for this task is usually the *mean square error* (MSE) [27] (also known as L_2 loss),

Figure 2.4: An example of image classification. When classified with ResNet-101 (pre-trained on ImageNet), the top-5 labels for this image are "tiger_cat, Egyptian_-cat, tabby, lynx, Siamese_cat".

defined in Equation (2.19), where y is the model output and t is the annotated value.

$$l(\mathbf{y}, \mathbf{t}) = \|\mathbf{y} - \mathbf{t}\|^2 \tag{2.19}$$

The CNN-based object detection models that will be seen in Section 2.3 must solve a multi-objective task that involves both classification and regression. There are many approaches to optimize multi-objective problems, but in the implementation of this work the losses with respect to both classification and regression are merged by addition.

2.2 Image Classification

Image classification is the task of mapping an image into one of a given set of categories. This is usually done by assigning scores for each category that should be high for the correct category and low for the incorrect ones. Figure 2.4 shows an example of image classification done by a ResNet-101 classifier.

With the success of the AlexNet [31] pioneer work, many deep CNN architectures were developed for computer vision tasks. On the image recognition end, works include GoogLeNet [38], Network in Network [39], VGG (Visual Geometry Group) [21], ResNet (Residual Network) [22], DenseNet (Densely Connected Convolutional Network) [40], ResNeXt [41], Inception (v2 [37], v3 [42], v4, -ResNet v1 and -ResNet v2 [43]), and MobileNets [44, 45]. These networks are trained in image classification datasets such as ImageNet [46].

Performance for image classification is usually measured by the accuracy of the

top-5 or top-1 results in a test set. Here, top-n means that if any of the best n classes scored by the network is correct it counts as a true positive.

2.2.1 VGG

VGG [21] is a deep convolutional neural network designed for the task of image classification. It is a widely employed CNN architecture due to its simple design.

It distinguishes itself from previous approaches (such as AlexNet), by using filters with small, 3×3, receptive fields, allowing the network to produce deep features with fewer weights.

Architecture

VGG comprises blocks of convolutional layers, all with the same number of output channels, kernel size 3×3, stride 1 and 1 zero-padding, such that the spatial size of feature maps within the same block is constant. Each of these layers also use ReLU activation function as a non-linearity. Between each block a max pooling layer, with 2×2 window size and 2 stride, is used to reduce the spatial dimension of the feature maps, moreover the first convolution of the next block doubles the number of channels. The last convolutional block is followed by three fully convolutional layers, two hidden with 4096 channels, and the last layer with a softmax for the classification outputs.

Dropout and weight decay are used as regularization methods when training VGG.

In this work the VGG-16 is used as a feature extractor for the object detection task, as will be seen in the following sections. Table 2.1 shows the structure of the layers of VGG-16. It is common to designate the n-th pooling layer (counting from the input to the output) of a network as $pool_n$, this is used in this work to identify the last pooling layer of VGG-16, $pool_5$.

Table 2.1: Configuration of the layers of VGG-16. All convolutions are followed by ReLU and have zero-padding 1, such that the input and output dimensions are the same at each stage. The hidden fully connected layers also have ReLU activation. Input dimensions have the channels omitted.

Input dimensions	Layer	Filter size	Stride	Number of filters
224×224	Convolutional $\times 2$	3×3	1	64
224×224	Max pooling	2×2	2	
112×112	Convolutional $\times 2$	3×3	1	128
112×112	Max pooling	2×2	2	
56×56	Convolutional $\times 3$	3×3	1	256
56×56	Max pooling	2×2	2	
28×28	Convolutional $\times 3$	3×3	1	512
28×28	Max pooling	2×2	2	
14×14	Convolutional $\times 3$	3×3	1	512
14×14	Max pooling	2×2	2	
25088	Fully connected 4096-d			
4096	Fully connected 4096-d			
4096	Fully connected 1000-d			
1000	Softmax			

2.2.2 ResNet

ResNet [22] is another deep CNN architecture that, at the time of its publication, achieved state-of-the-art results in the ILSVRC [47] classification task. ResNet also served as a basis for many other CNN systems such as ResNeXt [41] and Inception-ResNet [43].

Other works exhibited problems dealing with optimization on very deep models. These problems were not caused by overfitting, since it was observed that deeper models solutions had worse performance not only on testing but also on training. However, deeper models should not produce worse results than shallower ones, since replacing some layers from them with identity mappings should replicate the shallow model. ResNet proposes a solution to creating deep models that can more easily replicate shallower ones, by adding an identity connection from feature maps from lower layers to higher ones.

In ResNet, connections, like the ones in Figure 2.5, are put between outputs of different layers. Note that, this way, the total output of the block can be defined as $H(x) = F(x) + x$, then the output of the convolutional layers is $F(x) = H(x) - x$. The idea is that the network is able to replicate a shallower one more easily, by making the feature maps computed by the convolutional layers close to 0 where

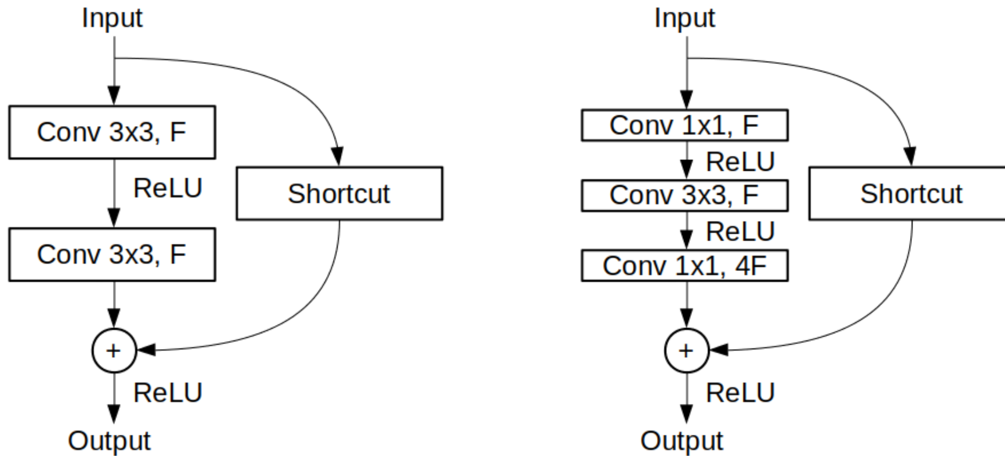


Figure 2.5: Building blocks of ResNet. The building block in the left is the one used in ResNet-18/34. The bottleneck building block in the right is used for ResNet-50/101/152. If the input has $4F$ channels and the same spatial dimension as the output the shortcut connection is the identity, otherwise the shortcut connection is a 1×1 convolutional layer that projects the input to the correct dimensions.

needed, as opposed to an identity map.

These connections are straightforward to implement when the dimensions are the same, in which case the feature maps are joined with element-wise addition. When spatial or channel dimensions are different, the identity mapping is replaced with 1×1 convolutional layers with stride and number of filters defined so as to match the input feature map dimensions with the output dimensions.

2.2.3 Architecture

ResNet presents a structure similar to VGG: most convolutions are 3×3 with 1 zero-padding, the number of filters of the 3×3 convolutions in each stage is the same and transitions between stages occur by reducing the number of spatial dimensions by a scale of two and doubling the number of channels. Stages are composed by building blocks, each block consisting of a sequence of convolutional layers with the input of the first layer connected to the output of the last one. Examples are shown in Figure 2.5[2]. The first convolutional layer uses 7×7 filters with stride 2, downsampling between stages is done with convolutions of stride 2 instead of max pooling layers (except for the second stage, that uses a max pooling layer of size 3×3 and stride 2), and the final layers are a global average pooling layer followed by one fully connected classification layer with softmax. Batch normalization is used after each convolutional layer, before the ReLU activation, and is applied after the connection in the last layer of the block. A weight decay of 0.0001 is used for regularization, but there is no dropout.

[2]Adapted from [22].

For the deeper models the blocks use a bottleneck architecture, composed of 1×1, 3×3 and 1×1 convolutional layers, instead of two 3×3 convolutions. The 1×1 convolutions reduce and increase the channel numbers, to let the 3×3 convolution operate on lower dimensions. Figure 2.5 presents a bottleneck building block and another without a bottleneck but with similar time complexity.

The object detectors implemented in this work are also trained and tested with the ResNet-101. Table 2.2 presents the stage and block structure of this network.

Table 2.2: Configuration of the layers of ResNet-101. All convolutions are followed by ReLU and have padding $\lfloor \frac{F}{2} \rfloor$ where F is the filter size, such that the output is the same size as the input when stride is 1. Stage k is denoted as conv$k$ (or Ck) and the first layer of each stage has stride 2. There is a shortcut connection between the input and output of every convolutional block.

Layer name	Output size	Layer	Filter size	Number of filters
conv1	112×112	Convolutional	7×7	64
conv2_x	56×56	Max pooling, stride 2	3×3	
		Conv block $\times3$	$\begin{bmatrix} 1\times1 \\ 3\times3 \\ 1\times1 \end{bmatrix}$	$\begin{bmatrix} 64 \\ 64 \\ 256 \end{bmatrix}$
conv3_x	28×28	Conv block $\times4$	$\begin{bmatrix} 1\times1 \\ 3\times3 \\ 1\times1 \end{bmatrix}$	$\begin{bmatrix} 128 \\ 128 \\ 512 \end{bmatrix}$
conv4_x	14×14	Conv block $\times23$	$\begin{bmatrix} 1\times1 \\ 3\times3 \\ 1\times1 \end{bmatrix}$	$\begin{bmatrix} 256 \\ 256 \\ 1024 \end{bmatrix}$
conv5_x	7×7	Conv block $\times3$	$\begin{bmatrix} 1\times1 \\ 3\times3 \\ 1\times1 \end{bmatrix}$	$\begin{bmatrix} 512 \\ 512 \\ 2048 \end{bmatrix}$
	1×1	Average pooling	7×7	
	1000	Fully connected 1000-d		
	1000	Softmax		

2.3 Object Detection

The task of object detection comprises locating and classifying objects in an image, by giving a bounding box of the position of the object and the class it belongs to, along with a score indicating the confidence of the detection, from a given set of classes. Figure 2.6 shows an annotated image and the result of object detection performed by an FPN detector.

Figure 2.6: An example of object detection with FPN on an image from the COCO validation set. The top image shows all ground-truth annotations. The bottom image shows the detections made by FPN with their respective confidences. Detections were filtered with a threshold of 0.5.

There are many approaches to this problem, mostly consisting of some variation of CNN, including R-CNN [4], Fast R-CNN [6], Faster R-CNN [7], R-FCN [13], FPN [8], YOLO [11, 12], SSD [9], DSSD [10] and RetinaNet [14]. Older methods used hand-engineered features such as SIFT [15, 16], HOG [17, 18] and Haar [19].

The CNN methods keep improving overtime. Improvements on CNN architectures affect all of these systems, moreover implementation details vary, so comparison between them is not an easy task. A systematic study by G-RMI (Google research and machine intelligence) [48], that compares Faster R-CNN, R-FCN and SSD, all reimplemented in TensorFlow, indicates that Faster R-CNN can achieve higher precision than the other two models, while SSD has the best precision when inference time is very limited, and is thus more geared towards real-time systems.

These detectors can be divided into two groups: one-stage and two-stage object detectors. One-stage detectors perform all the detections with one pass of the CNN, generating all classification and regression outputs directly. These include YOLO and SSD, that can achieve very low inference time with fairly good accuracy, but they do not handle objects of small size well (although this seems to be less of an issue in DSSD and RetinaNet). On the other hand, Faster R-CNN and R-FCN are two-stage detectors, that is, they first generate region proposals and then classify those regions.

2.3.1 Intersection over Union

Intersection over union (IoU) [25] is a metric that compares two bounding boxes by calculating the ratio of the bounding boxes' intersection area over the bounding boxes' union area. It is a useful metric to determine if a detection has enough overlap to be matched with a ground-truth object from a labeled dataset. The PASCAL VOC [25, 26] and the COCO [24] detection tasks utilize it to identify true positives and false positives from each class by comparing the maximum IoU, among the IoU values obtained by comparing with every ground-truth bounding box from that class, of each detection with a threshold.

2.3.2 Mean Average Precision

Informally, a good object detector can localize objects well, classifies them precisely and misses few detections of the objects of the desired classes. The first characteristic is measured with IoU, the second is called precision and the third is known as recall, and all of these are related to the concept of true positives and false positives in object detection.

A true positive (TP) is a detection that correctly locates an object in the image and identifies the category. To identify a TP in labeled data, the detected bounding

box is compared with all unmatched ground-truths of the detected class using the IoU. If any of those is greater than a threshold, this detection is considered a TP and is matched to the groud-truth object it has the highest overlap with. Note that multiple detections of the same object are undesired, since the detections are assumed to be from different instances, so detections are matched to ground-truths in order of score. All detections that remain unmatched are considered false positives (FP).

Then, the precision and recall of the detector are defined as in Equations (2.20) and (2.21). Note that, since a higher confidence score indicates that the detection is more likely to be a true positive, a threshold can be used to filter the detections with low scores. This threshold usually increases the precision, but decreases the recall in trade.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.20}$$

$$\text{Recall} = \frac{\text{TP}}{\text{Groud-truth positives}} \tag{2.21}$$

Taking into account those considerations, one of the metrics commonly employed to measure performance of an object detector is the *mean average precision* (mAP). The mean average precision is the mean value of the average precision computed over every class. The average precision of a class is the area under the interpolated precision / recall curve.

To build the precision / recall curve, a pair of precision and recall values of the object detector are computed at every possible score threshold. The curve is formed by the interpolation of all these pairs in the Cartesian plane. Note that, since the number of detections is finite, only a finite number of thresholds need to be used, that is, those that are equal to the confidences of each of the detections. Furthermore, this function is modified to become non-increasing, with respect to recall, by substituting a precision of a given recall by the maximum precision with recall values equal to or larger than it. An example of a precision / recall curve and its interpolated version are illustrated in Figure 2.7.

2.3.3 R-CNN

With the breakthrough of CNNs in the image classification task, many detectors arose using these classifiers as a base. *Regions with CNN features* (R-CNN) is one such approach, using the CNN features to classify the rectangular proposal regions. At the time of the original publication, R-CNN was shown to achieve state-of-the-art performance on the PASCAL VOC 2012 dataset with an mAP of 62% with VGG-16.

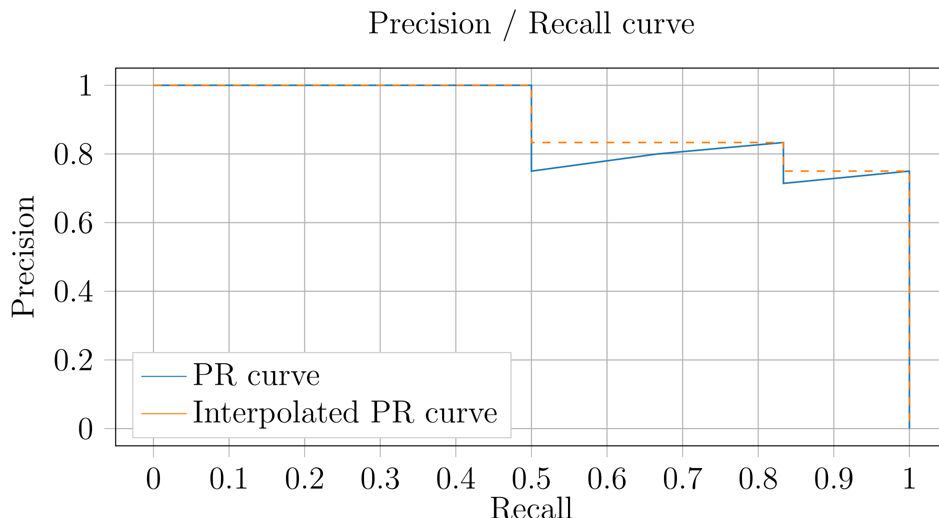


Figure 2.7: Example of a precision / recall curve using the detection vector (ordered by score): [T, T, T, F, T, T, F, T, F, F], where T is true positive and F is false positive, with a total of 6 ground-truths.

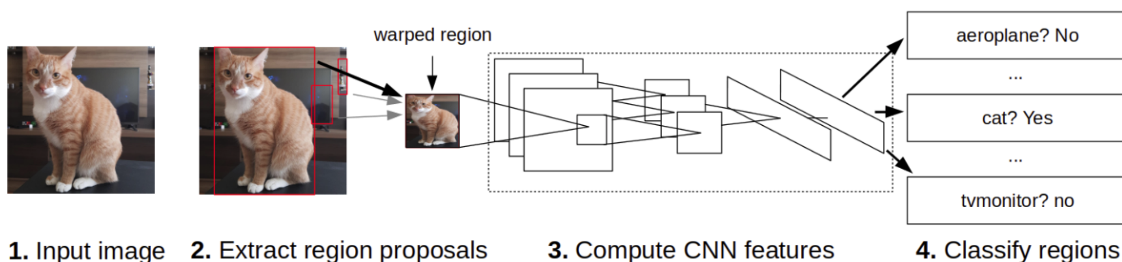


Figure 2.8: Illustration of the R-CNN architecture. (1) loads an image; (2) around 2000 proposals are generated by selective search and warped to a fixed size, to be processed by a CNN; (3) features are extracted using a CNN; (4) the system classifies the features with binary SVMs (one per class)

R-CNN [4] can be summarized as follows: extract some region proposals from an input image, compute the features of those regions with a CNN, and classify these features with a support vector machine (SVM), as can be seen in Figure 2.8[3].

Modules

The first module of the method consists of a class agnostic region proposal method. The region proposals are defined by bounding boxes. Originally the selective search algorithm [49] was chosen for the R-CNN system, but other methods can be employed. Around 2000 proposals are generated per image, but most of them will be discarded on classification.

The second module consists of a CNN feature extractor. This network can be obtained by taking a pre-trained classifier (originally, VGG-16) and removing the

[3] Adapted from [4].

fully connected layer. The CNN is responsible for extracting the features from the region proposals. The regions must first be transformed, to fit the dimensions expected by the CNN, to attend this the pixels in that region are warped (a margin of 16 pixels in the warped region is also given, for context).

The third module does classification and regression on the regions, based on the extracted feature vectors. Here, for each class, a linear SVM is used to compute the score of the proposals. Furthermore, to avoid multiple detections of the same object, a greedy *non-maximum suppression* (NMS) is performed on each class independently, based on the scoring given by the SVM.

This greedy NMS is used to filter bounding boxes of the same class that are too close to each other, with the idea that proposals with a higher score are more likely to be closer to the detected object. Given a list of proposal bounding boxes from a class, the algorithm works as follows: the bounding boxes are ordered using their corresponding scores; the IoU values between the highest scoring bounding box in the list and all other bounding boxes is computed; the bounding boxes which are too close, i.e. have an IoU above a certain learned threshold (0.3), with the highest scoring bounding box are removed from the list; this selected proposal is included in the list of post-NMS proposals; this process is repeated on the remaining proposals in the pre-NMS list. This process guarantees that, for each class, all the bounding boxes remaining does not have an overlap with each other greater than the threshold used.

Since the region proposals returned by the first module may be misaligned with the object supposed to be detected, the coordinates of these proposals are adjusted, depending on their features and classifications. The coordinates of a proposal are given by the 4-tuple $\mathbf{P} = (P_x, P_y, P_w, P_h)$, where (P_x, P_y) is the coordinate pair for the center pixel of the bounding box, and P_w and P_h are its width and height in pixels. The adjustments are made as in Equations (2.22), (2.23), (2.24) and (2.25), where $\mathbf{A} = (A_x, A_y, A_w, A_h)$ is the new coordinate 4-tuple and $\mathbf{t} = (t_x, t_y, t_w, t_h)$ is a vector that is computed as a linear function of features of the region proposal (in the original work, these features are taken from the $pool_5$ layer of the CNN). This adjustment is done after classification and is class-specific.

$$A_x = P_x + P_w t_x \tag{2.22}$$

$$A_y = P_y + P_h t_y \tag{2.23}$$

$$A_w = P_w \exp(t_w) \tag{2.24}$$

$$A_h = P_h \exp(t_h) \tag{2.25}$$

Training

To obtain the second module, a CNN, pre-trained on ImageNet [46], is fine-tuned to adapt to the object detection task and the warped inputs. The original classification layer is replaced by a randomly initialized layer with the same dimensions as the number of classes (plus background) of the dataset that will be used for fine-tuning. If a proposal has IoU higher than 0.5 with some ground-truth, it is considered as being from the class of the ground-truth that has the highest overlap with it, otherwise it is considered as background. To optimize, SGD with learning rate 0.001 and mini-batches of size 128 (32 class proposals and 96 background proposals) is used.

The classification layer used for fine-tuning is then replaced by the linear SVM classifiers. To train these classifiers, proposals with IoU lower than 0.3 are considered negatives, while ground-truth bounding boxes are considered positives. The SVM from each class is then trained with hard negative mining [4, 18], that is to search for examples that are likely to generate false positives to collect a small number of hard negative samples. This is done to limit the size of the training data and to balance the classes, since background is more common.

Finally, regression is performed by optimizing the weights of the linear layer that generates t with an L_2 regularized least squares objective. The samples used in this regression are all proposals with at least 0.6 IoU with some ground-truth bounding box, and the targets are the values of t that map to the coordinates of the nearest ground-truth box. The minimum overlap requirement is important to allow convergence, by ensuring that the features contain some information about the ground-truth region.

2.3.4 Fast R-CNN

An issue of the R-CNN method is that its inference takes around 47 seconds per image, which is too long for some applications. Most of this time is spent on feature extraction, since R-CNN forward iterations the CNN over each region proposal. Training is also slow and requires three stages (CNN fine tuning, SVM training and bounding box regression). Fast R-CNN [6] is an improvement upon R-CNN that achieves much faster inference and training time as well as a higher mAP of 66% (with VGG-16) on the PASCAL VOC 2012 challenge. An overview of Fast R-CNN can be seen in Figure 2.9[4].

Fast R-CNN also comprises three modules, with the first one being identical to the one from R-CNN. The second module is a CNN that extracts feature maps (3-D tensors). The third module is an ANN that classifies the region proposals projected

[4]Adapted from [6].

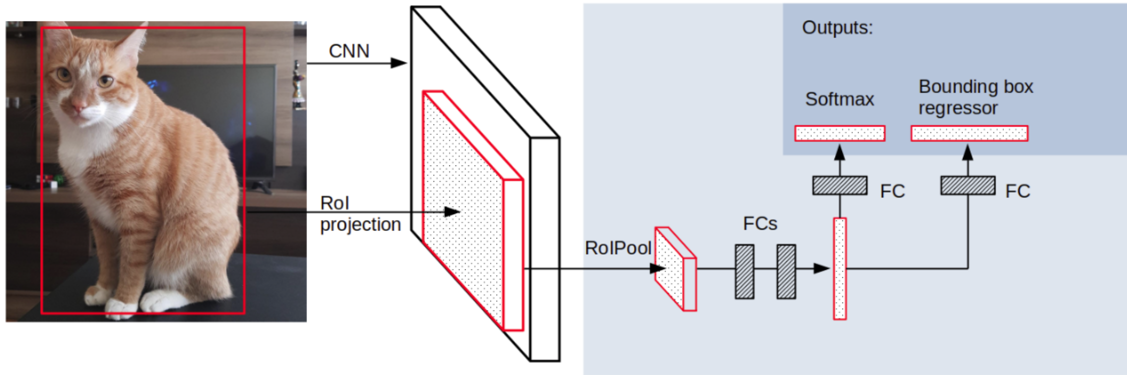


Figure 2.9: Illustration of the Fast R-CNN architecture. Unlike R-CNN, the features of the convolutional layers are extracted once, then the regions proposed by selective search are projected to the feature maps of the last layer. These projected regions are then passed to the region-specific fully connected layers, for classification and regression, by the RoI pooling layer. The last module is applied once per region of interest, but the process is made much lighter by removing the need to call the full CNN on each proposal.

on the features of the second module. The second module is only called once per image, speeding up the inference process. Only the classification module needs to be run multiple times.

The second and third module are derived from a pre-trained CNN designed for classification. A pooling layer ($pool_5$ for VGG-16) from this network is replaced by a region of interest (RoI) pooling layer (that pools regions of the feature maps to the classification module). Furthermore, the last fully connected layer is replaced by two other randomly initialized fully connected layers. One of them is followed by a softmax that performs classification over all object classes and a background class, while the other is a layer that performs adjustments on the bounding boxes according to the class.

The original region proposals are projected to the feature maps by multiplying all coordinates in the region by the inverse of the total stride (16 for VGG-16) of those feature maps. These projections are quantified, i.e. the divisions by the stride are rounded, so that the region of interest in the feature maps is properly defined.

RoI pooling layer

The RoI pooling layer converts, with max pooling, all proposal regions in the feature map to the spatial dimensions required by the classification stage (7×7 for VGG-16). If the input has spatial dimensions $w \times h$ and the output requires dimensions $W \times H$, RoI pooling will approximate a max pooling layer with size and stride $\frac{w}{W} \times \frac{h}{H}$. Since these divisions can leave a remainder it is possible that the windows of this pooling are uneven. An example can be seen in Figure 2.10, where a region of interest is

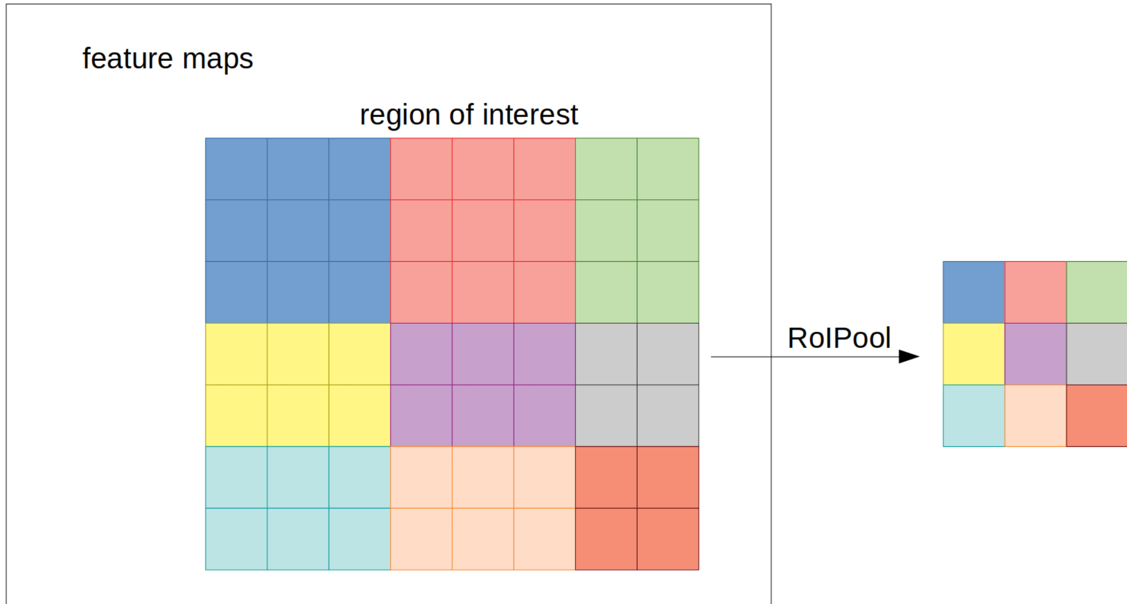


Figure 2.10: A region of interest of dimensions 7×8 in the feature maps is extracted with RoIPool to 3×3 feature maps. Each colored window represents one value in the output. The output is determined by taking the maximum inside each window, for each channel independently.

pooled from dimensions 7×8 to 3×3.

Loss function

Fast R-CNN optimizes classification and the bounding box regressors jointly. To achieve this, a multi-objective loss L is defined as in Equation (2.26), where $\mathbf{p}$ is the class probability vector computed by the classification layer, c' is the ground-truth class, $\mathbf{t}^{c'}$ is the regression 4-tuple for class c' computed by the regression layer and $\mathbf{t}'$ is its ground-truth. The ground-truth is the closest object to the proposal that satisfies a minimum overlap.

$$L(\mathbf{p}, \mathbf{t}^{c'}, c', \mathbf{t}') = L_{cls}(\mathbf{p}, c') + L_{reg}(\mathbf{t}^{c'}, \mathbf{t}', c') \tag{2.26}$$

The loss with respect to classification is the log loss function $L_{cls}(\mathbf{p}, c') = -\log(p_{c'})$. With respect to regression, the loss is zero when c' is background and is the smooth L_1 function otherwise. This regression loss is described in Equations (2.27) and (2.28). This loss function is chosen to make the regressor less sensitive to outliers in comparison to the L_2 loss employed in R-CNN.

$$L_{reg}(\mathbf{t}^{c'}, \mathbf{t}', c') = \sum_{i \in \{x,y,w,h\}} \text{smooth}_{L_1}(t_i^{c'} - t_i') \tag{2.27}$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{2.28}$$

Training

Fast R-CNN is trained with mini-batches of size 128, composed of 32 object classes and 96 background regions. Object class regions are taken from proposals with IoU greater than 0.5 for some ground-truth bounding box, while background regions have IoU in the range $[0.1, 0.5)$ with its most overlapping ground truth. Weights are optimized with SGD with learning rate 0.001 (decaying to 0.0001 after some iterations) and weight decay 0.0005.

This method also offers a major speedup over R-CNN while training. The mini-batches are taken from a few images (2 in the original [6]), thus the number of CNN calls is reduced, when compared to the 128 required in optimizing R-CNN. While this could cause slow training convergence, since region proposals from the same image are more likely to be highly correlated, it was observed that this concern does not hold an issue in practice.

2.3.5 Faster R-CNN

Fast R-CNN solved the bottleneck R-CNN had on the number of forward iterations performed on the CNN, but it highlighted a new issue. Most of the time spent on inference on Fast R-CNN is due to selective search, the proposal generation algorithm. Faster R-CNN [7] reduces this time by embedding the region proposal generation in the network. This method for generating proposals with CNN is called *region proposal network* (RPN).

RPN

RPN is a CNN that identifies where objects might be present in the input image. It is a class-agnostic object detector that, in the Faster R-CNN architecture, substitutes selective search as the region proposal generation method. In Faster R-CNN the RPN shares the feature extractor layers with the Fast R-CNN architecture, branching out at the feature map where the regions of interest are pooled, such as in Figure 2.11[5].

The region proposals are based on fixed bounding boxes denominated anchors. Every spatial position in the feature map is matched with a set of k anchors, of different sizes and proportions, that are centered in the corresponding pixel of the image. The whole set of anchors can be visualized as bounding boxes centered in a

[5]Adapted from [7].

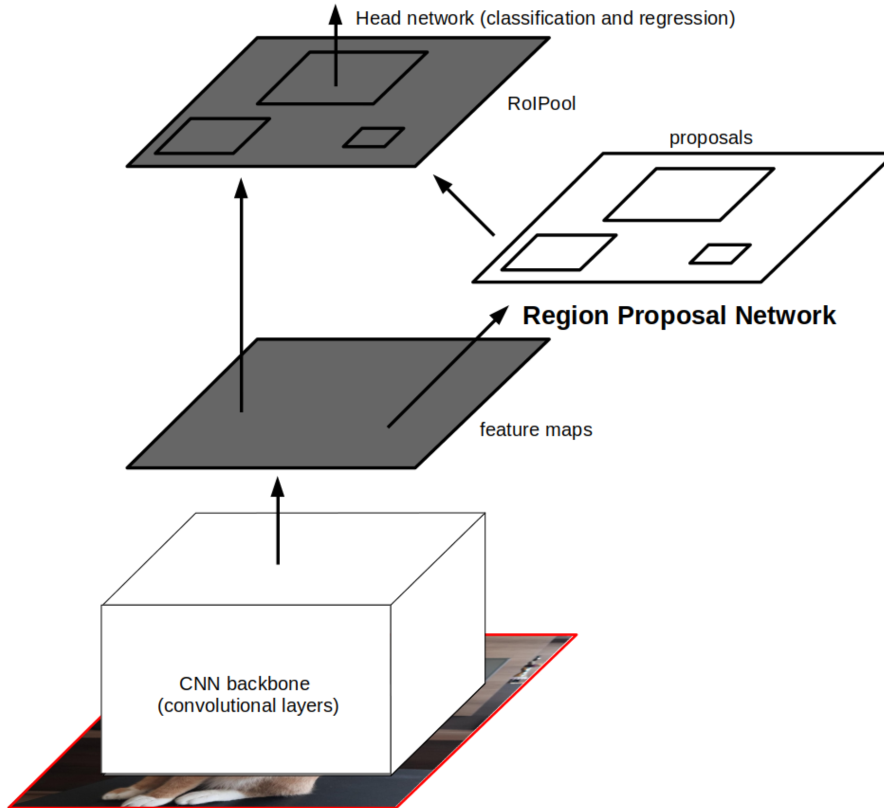


Figure 2.11: Illustration of the Faster R-CNN architecture. In Faster R-CNN the proposal generation module (RPN) shares layers with a Fast R-CNN system.

grid of pixels, with spacing equal to the network stride (the stride accumulated over all preceding layers of the network), over the input image. Examples of anchors can be seen in Figure 2.12[6].

RPN generates two output 3-D tensors, one for anchor classification and another for bounding box regression. Each spatial coordinate in the anchor classification tensor is a vector that contains $2k$ values that comprises the probabilities of each of the k anchors in that position being an object or not. Each of these vectors has a sibling vector of $4k$ values comprising 4-tuples that are used to adjust the anchors with the same bounding box transformation method described in Equations (2.22), (2.23), (2.24) and (2.25).

The outputs are computed by convolutional layers connected to the base feature extractor CNN. The feature map of the extractor is passed to a convolutional layer with filter size 3×3 and 512 filters, followed by ReLU. This creates one 512-D feature vector per position, that is then used to compute the values for classification and regression, for the anchors centered in that position, through two sibling linear layers.

[6] Adapted from [7].

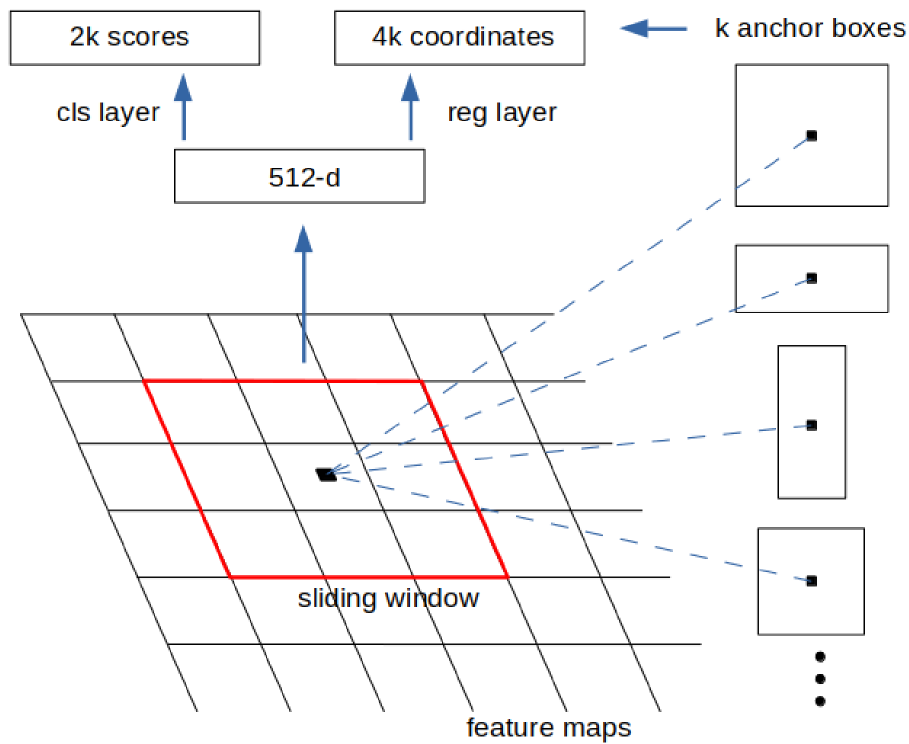


Figure 2.12: Overview of the RPN specific layers. A sliding window scans the feature maps to generate features related to specific positions at the image. These features are scored with a fully connected layer to define how likely it is that a bounding box of specific size (an anchor) placed at that position fit well an object, for each of the k pre-defined anchors. A sibling layer for regression is also used for each of the anchors. These extra layers are implemented with a fully convolutional network using a 3×3 convolution followed by ReLU and two sibling 1×1 convolutions for classification and regression.

Training RPN

Definitions of positive and negative (background) anchors are required to train this new network. A positive anchor is one that either has an IoU of at least 0.7 with any ground-truth bounding box or has the highest IoU with a ground-truth bounding box. A negative anchor is a non-positive anchor that has less than 0.3 IoU with all ground-truth bounding boxes. Note that some anchors are neither and therefore do not contribute to training.

The loss function for an anchor sample in RPN is the same as the one defined for proposal classification and regression in Fast R-CNN, with the only difference being that the classification output and target are binary.

Training RPN requires some care, since using all anchors will bias the network towards background. Taking this into account, 256 anchors are sampled per mini-batch: 128 for object proposals and 128 for background proposals. If the images in the mini-batch do not have 128 positive samples, the mini-batch is completed with background samples.

Training Faster R-CNN

Since Fast R-CNN and RPN share some layers, a strategy needs to be developed to train the architecture for both. In the original article three methods are proposed.

The first one is alternating training. In this solution, the RPN is trained first, and its proposals are used to train a separate Fast R-CNN. The layers from Fast R-CNN are then shared, the shared weights are fixed, and fine-tuning is performed on the weights exclusive to RPN. Then, with the shared weights still fixed, the network weights exclusive to Fast R-CNN are fine-tuned.

The second solution is approximate joint training. Proposals from RPN are treated as fixed, with respect to Fast R-CNN training, and the loss functions of RPN and Fast R-CNN are merged by addition, so that both methods are optimized jointly. This method was found, empirically, to produce results similar to the first one, but reducing training time. This solution is the one used in this work.

The third solution is non-approximate joint training. This one does not treat proposals as fixed, and thus gradients from the pooling layer outputs flow to the RPN layers. Note that this requires a differentiable RoI downsampling method, instead of RoI pooling, such as RoIAlign [3].

Training is done with SGD with hyperparameters similar to those from Fast R-CNN. The hyperparameters used in this implementation will be discussed in Chapter 3.

2.3.6 Feature Pyramid Networks

While the anchor approach from RPN makes Faster R-CNN more robust to translated objects, detecting objects with different scales in an image is still a challenge. Most solutions [5, 22, 50] are based on multi-scale methods that extract sets of feature maps of multiple scales, called *feature pyramids*, from sets of images of multiple scales, called *image pyramids*. These methods achieve partial scale invariance because objects in the image will have their features extracted from a range of scales, so, scaled objects can be represented by similar features by shifting the level in the feature pyramid.

These methods achieved good results and could be used to improve other architectures. But calculating the features at multiple scales is computationally demanding, moreover algorithms for training image pyramids end-to-end are impractical because of memory limitations, making them only useful to improve performance on inference.

Feature pyramid network (FPN) [8] is a solution that deals with these issues by generating a feature pyramid from a forward pass on a CNN in a single image scale. This feature pyramid is constructed from the feature maps from multiple layers of the CNN. A concern with directly using the feature maps computed with a CNN as a feature pyramid is that the features from the lower layers are shallow and may be difficult to be used to effectively detect objects. To solve this, FPN strengthens the features from lower layers with context from the upper layer. This way, the lower layers can have strong semantic information, aside from the more localized features.

Figure 2.13[7] shows the different options, that were discussed here, to predict with a CNN, including the classic way using a single convolutional layer output and the models that build feature pyramids.

FPN architecture

A given CNN topology can be transformed in an FPN by including a few small layers to make top-down connections from the deepest feature map to the lower ones, and lateral connections from the lower features maps adding into this downstream. Figure 2.14[8] illustrates the transformed topology.

The CNN generates many feature maps from the bottom-up pass. They can be separated in stages, where the features maps' spatial dimensions do not change. The feature pyramid is constructed with one feature from each stage. Since deeper layers are expected to have stronger features, the pyramid levels are built upon the outputs from the deepest layer of each stage. The feature maps from the i-th pyramid level

[7]Adapted from [8].
[8]Adapted from [8].

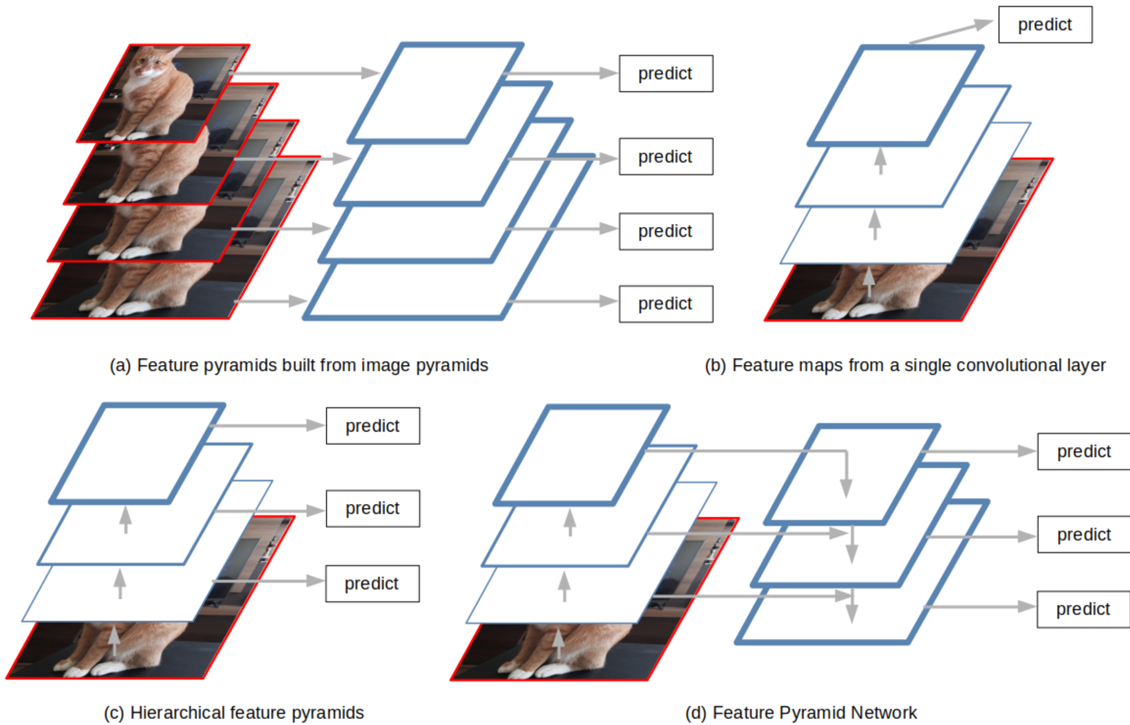


Figure 2.13: Different prediction models for object detection: (a) Prediction is done from a feature pyramid built upon an image pyramid. (b) Classic CNN, prediction is calculated from the feature maps from only one convolutional layer. (c) Feature pyramid prediction, but the feature pyramid is built using the feature hierarchy directly. (d) An improvement of (c) by including top-down and lateral connections to get features with strong context in every level. It is also much less computationally expensive than (a), having inference speed closer to (b) and (c) while having better performance.

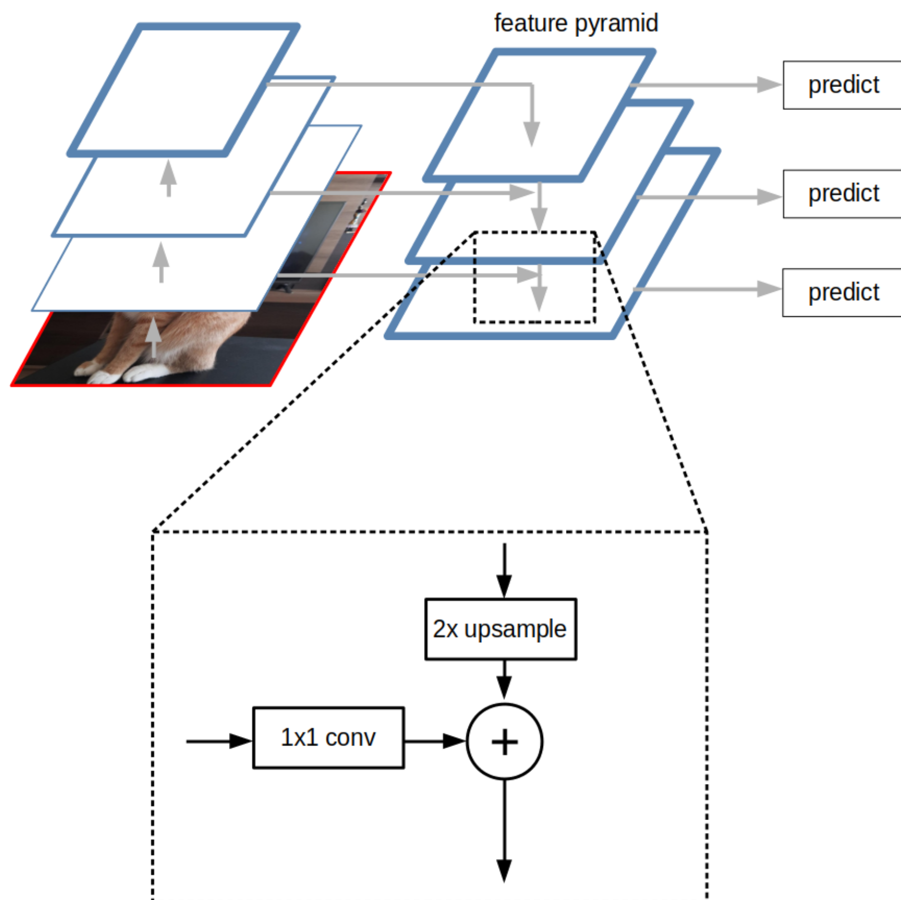


Figure 2.14: Layers introduced by the FPN. The lateral connection is a 1×1 convolutional layer that equalizes the number of channels at all pyramid levels, while the top-down connection is a $2\times$ upsample that introduces deeper semantic information into the lower levels.

will be denoted as C_i.

The top-down layers upsample feature maps, from their respective pyramid level, and pass them down to be added with feature maps of the pyramid level directly below it. The lateral connections are 1×1 convolutional layers that map the bottom-up features to a fixed channel size d. These features are then added to the downstream features. Finally, 3×3 convolutional layers are applied to the combined features to reduce the effect of aliasing from the upsampling.

The above description can be systematized as follows. Let S be a set of integers corresponding to the pyramid levels, $(C)_{i \in S}$ be the bottom-up feature pyramid, $(M)_{i \in S}$ be the merged feature pyramid, and $(P)_{i \in S}$ be the final feature pyramid, then $M_{\max S} = \text{conv}_{1\times 1}(C_{\max S})$, and, for $i \neq \max S$, $M_i = \text{conv}_{1\times 1}(C_i) + \text{upsample}(M_{i+1})$. Finally, $P_i = \text{conv}_{3\times 3}(M_i), \forall i \in S$. Here $\text{conv}_{1\times 1}$ denotes a convolutional layer of kernel size 1, stride 1 and d output channels, while $\text{conv}_{3\times 3}$ denotes a convolutional layer of kernel size 3, stride 1, padding 1 and d output channels.

FPN as an RPN

An RPN can be defined from an FPN with minor changes. The 3×3 convolutional layer and the sibling 1×1 convolutional layers used for classification and regression are applied to all levels of the feature pyramid. Also, each pyramid level is assigned to find object proposals of a size proportional to its level. This lets each level be assigned only one anchor scale (aspect ratios remain the same).

To train this network, the anchors are labelled exactly as defined in the original RPN, using IoU values with ground-truth bounding boxes to determine thresholds for negative and positive. However, depending on the scale of the anchors sampled for training, the gradient will flow through different levels of the pyramid.

The weights of the convolutions operating at each pyramid level are shared. As noted in the FPN paper, this produces similar results to the alternative without sharing parameters.

Fast R-CNN with FPN

To build a Fast R-CNN with FPN, it is necessary to define how to assign a region of interest to a pyramid level. In SPP-net (spatial pyramid pooling) [5] and Fast R-CNN the multi-scale inference approach builds a feature pyramid from an image pyramid, then each region of interest is assigned to the level where its scale (in the image) is closer to the CNN's expected input scale for classification (usually an area of 224^2). A solution for FPN is to use this method, assuming the feature pyramid was built from an image pyramid, and choosing a default level k_0 which is used to infer from proposals of scale 224. This process assigns a region of interest of width

w and height h to a pyramid level k as described in Equation (2.29).

$$k = \lfloor k_0 + \log_2(\sqrt{wh}/224) \rfloor \tag{2.29}$$

Finally, all the proposals are reshaped, with methods such as RoI pooling, and a network for classification and regression is run through every reshaped region of interest.

Taking into consideration the notes above, a Faster R-CNN architecture based on FPN can then be trained with the same methods previously discussed.

2.3.7 Mask R-CNN

The architectures that were discussed above are flexible, being able to solve tasks, other than object detection, with the inclusion of new modules. Mask R-CNN [3] is a system based on Faster R-CNN that makes predictions for the *instance segmentation* problem. The instance segmentation task is the problem of classifying and finding a pixel mask for all objects in an image (a similar problem is the *semantic segmentation* task, where pixel masks must be found for every class, disregarding the problem of distinguishing objects of the same class).

Mask R-CNN also improves object detection mAP, with ablation experiments showing that the instance segmentation specific module is partially responsible for this improvement. This means that including annotated segmentation data introduces useful information for the object detection task and allows the system to learn better features. The RoIAlign layer is also introduced and is shown to produce better results than RoIPool in both object detection and instance segmentation.

To build a Mask R-CNN architecture a *fully convolutional network* (FCN) [23] is appended to the *head network* (the network that runs on top of the regions of interest). This FCN predicts a small $m \times m$ mask (each entry is the output of a sigmoid) for each class on each object instance. When the class of a proposal is determined by the classification output, the mask of the given class is then resized to the original size of the region of interest and transformed into a binary mask with a threshold of 0.5.

Further details are beyond the scope of this work and can be seen in the original paper [3]. But the RoIAlign operation is used in the implementation and is explained below.

RoIAlign

To redimension regions of interest using RoIPool, quantizations have to be done in two steps. When the proposal is projected to the feature maps, the coordinates must be divided by the stride and rounded, so they can be assigned to a region of

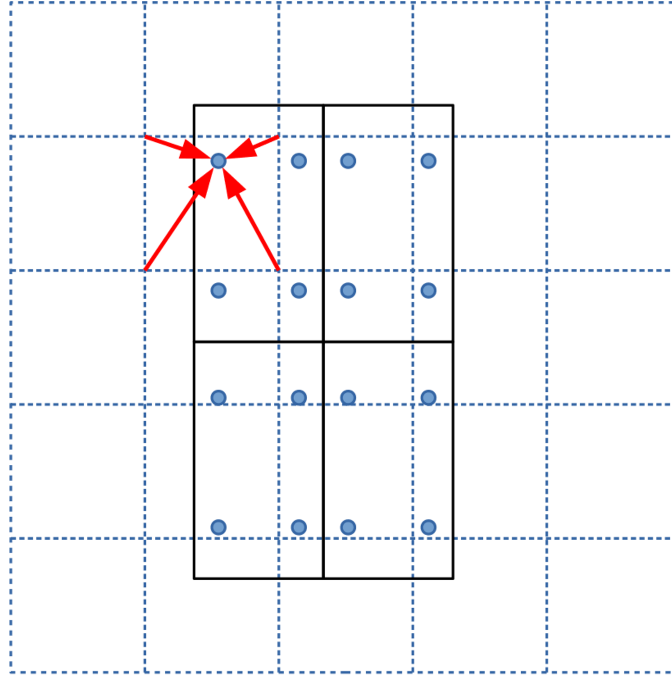


Figure 2.15: This figures illustrates the RoIAlign process for one region of interest. The dashed grid represents the feature maps, the solid grid represents a projected region of interest, where the rectangles are the bins, and the blue dots are the samples that are computed with bilinear interpolation.

the feature maps with defined values. The other quantization occurs in the RoIPool, when the widths and heights of the pooling sub-windows are approximated. These quantizations can be an issue when the task requires finding exact pixel positions of the objects, since those approximations can cause a misalignment on the pooled features and the original region of interest.

The RoIAlign layer is introduced to solve this alignment problem. In this method, no rounding is performed when projecting the regions of interest, furthermore, for each projected region, a grid of bins is defined, with the dimensions required by the head network. The values at sampled positions around each bin are calculated using bilinear interpolation in the feature maps. These values are then used to compute the value in the bin by taking the max or the average between them. Figure 2.15[9] illustrates this process.

Bilinear interpolation is the operation described in Equation (2.30), where (x, y) is a position within the bin, (x_0, y_0), (x_0, y_1), (x_1, y_0) and (x_1, y_1) are the points in the feature maps closest to the bin, with $x_0 \neq x_1$ and $y_0 \neq y_1$, f is a function that returns the values of the feature maps at that spatial location (which is a vector in the channel dimension), and $\tilde{f}$ is the function that returns the bilinear interpolation of f. Gradient calculation is straightforward, so no complications are introduced in optimization.

[9]Adapted from [3].

$$\tilde{f}(x,y) = \frac{(x_1 - x)(y_1 - y)f(x_0,y_0) + (x_1 - x)(y - y_0)f(x_0,y_1)}{(x_1 - x_0)(y_1 - y_0)} + \\ + \frac{(x - x_0)(y_1 - y)f(x_1,y_0) + (x - x_0)(y - y_0)f(x_1,y_1)}{(x_1 - x_0)(y_1 - y_0)} \tag{2.30}$$

This layer significantly boosts the performance of Mask R-CNN on instance segmentation. It also improves Faster R-CNN object detection results by around 1.1% mAP.

Chapter 3

Implementation

This chapter describes the implementation details of the Faster R-CNN and FPN detectors developed in this work. The framework utilized to implement the detectors is described in Section 3.1, the implementations of NMS and IoU are presented in Section 3.2, further details on the implementations of Faster R-CNN and FPN can be found in Sections 3.3 and 3.4, respectively. Finally, a summary of the hyper-parameters is presented in Section 3.5.

Among the techniques discussed in the previous chapter, Faster R-CNN, FPN, IoU, NMS, training and inference are explicitly implemented in the code. Average precision is implemented, but the results available in Chapter 4 are calculated with the official competition codes from a formatted file containing the results from the inference. Convolutional and linear layers, activation functions, loss functions, ResNet and VGG[1] implementations are obtained from the PyTorch library. A CUDA C implementation of RoIAlign is used in this work, obtained from another repository[2].

3.1 Framework

A PyTorch environment is utilized for the development of the detectors. PyTorch is a Python library that can be used to mount computational graphs, as described in Section 2.1.3. It comes with a variety of classes and functions that implements layers, activation, loss, data loading and processing, and optimization algorithms.

The pre-trained VGG-16 and ResNet-101 networks were taken from the PyTorch repository.

[1]The PyTorch VGG and ResNet implementations are available, respectively, in the links: `https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py` and `https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py`.

[2]`https://github.com/facebookresearch/maskrcnn-benchmark`.

3.2 Non-maximum suppression

While in other modules NMS is usually small and fast, in the RPN, NMS needs to be applied to around 21k proposals in Faster R-CNN and 60k in FPN. For this reason, efficient implementations of NMS, and therefore also IoU, are essential to obtain reasonable training and test times.

IoU is implemented as a function that receives two batches of bounding box coordinates. These batches are organized in matrices and the output IoU is a matrix in which the value at (i, j) is the IoU between the i-th bounding box in the first batch and the j-th bounding box in the second batch. This matrix is obtained with simple operations, so as to allow PyTorch to perform parallel computation in the GPU. Consider $(X_{\text{min}}, Y_{\text{min}})$, $(X_{\text{max}}, Y_{\text{max}})$ as the pair of vectors containing the top-left and bottom-right points of the bounding boxes in one batch respectively. For the first batch these vectors are column vectors, while for the second batch they are row vectors. Both vectors are extended into matrices of the same size, by repeating the rows for the row vectors and columns in the column vector. Equations (3.1) through (3.8) show how the IoU is computed using the subscripts 1, 2, I, to distinguish the first, second and intersection batches respectively. All operations are performed element-wise.

$$X_{\text{min}\,I} = \max(X_{\text{min}\,1}, X_{\text{min}\,2}) \tag{3.1}$$
$$X_{\text{max}\,I} = \min(X_{\text{max}\,1}, X_{\text{max}\,2}) \tag{3.2}$$
$$Y_{\text{min}\,I} = \max(Y_{\text{min}\,1}, Y_{\text{min}\,2}) \tag{3.3}$$
$$Y_{\text{max}\,I} = \min(Y_{\text{max}\,1}, Y_{\text{max}\,2}) \tag{3.4}$$
$$A_1 = (X_{\text{max}\,1} - X_{\text{min}\,1})(Y_{\text{max}\,1} - Y_{\text{min}\,1}) \tag{3.5}$$
$$A_2 = (X_{\text{max}\,2} - X_{\text{min}\,2})(Y_{\text{max}\,2} - Y_{\text{min}\,2}) \tag{3.6}$$
$$A_I = (X_{\text{max}\,I} - X_{\text{min}\,I})(Y_{\text{max}\,I} - Y_{\text{min}\,I}) \tag{3.7}$$
$$\text{IoU} = \frac{A_I}{A_1 + A_2 - A_I} \tag{3.8}$$

In NMS, the inputs are a matrix, containing the coordinates of the bounding boxes in each row, and a corresponding score vector. Using the score vector, the rows of the matrix are sorted such that the proposal at the top is the one with the highest score. Computing the complete IoU matrix between the RPN proposals is inviable due to memory limitations, so the matrix is computed in batches of 1000 rows, which are then compared to the 0.7 threshold and converted to a binary mask. Since the next step is procedural, the binary mask is passed to the CPU and used to filter the proposals.

The filtering process is done by iterating over the rows of the binary mask matrix from top to bottom, building a binary vector, with 0 corresponding to filtered proposals, and appending indices to a list of the remaining proposals. The binary vector is initialized as the first row of the matrix, and the list initially contains only 0, corresponding to the highest scoring proposal. On the i-th iteration, if the i-th proposal was not filtered (i.e. if the i-th value of the binary vector is 1), a binary "and" operation is done between the vector and the i-th row of the binary mask (filtering all proposals that have enough overlap with the i-th proposal), and i is appended to the list. Otherwise, if the proposal was already filtered, nothing happens in that iteration. After this process iterates over all proposals, the list of remaining regions of interest is complete, all other regions are discarded and NMS is done.

3.3 Faster R-CNN

The first operation performed by Faster R-CNN is to apply the backbone network on the input image to generate the feature maps used in the subsequent steps. Then, the RPN specific layers are used to generate the proposals, that are filtered by NMS with threshold 0.7. RoIAlign is applied to these filtered regions of interest, that are then classified and regressed by a head network. Therefore, to build this Faster R-CNN system, it is necessary to specify the design of the backbone and head layers, as well as the hyperparameters that define the architecture of the RPN, the thresholds of the NMS operations and the number of region proposals filtered.

The system is built by passing the backbone, head and configuration variables to the Faster R-CNN constructor.

3.3.1 Backbone

The backbone is the network that extracts the feature maps that are used to sample object-specific features for the Fast R-CNN head and proposals by the RPN. Backbones are usually taken from pre-trained image classification CNNs, since useful features for image classification and object detection are assumed to be similar. In this work, Faster R-CNN is implemented with two CNN backbones, VGG-16 and ResNet-101, both with pre-trained weights taken from the PyTorch repository.

The VGG-16 backbone corresponds to all the layers before and excluding $pool_5$, that is, all the convolutional layers. The ResNet-101 backbone comprises all layers up to the $C4$ stage. Experiments [50] using a ResNet backbone up to the $C5$ stage produce worse results, possibly because of the reduced spatial dimension of the feature maps from the last stage, but this is solved in FPN.

In VGG-16 the convolutional layers up to conv2_2 are kept frozen, that is, they

are not trained (according to the Fast R-CNN paper [6], this has little effect on mAP), whereas ResNet-101 has conv1 and the batch normalization layers frozen. This is done to save memory and to reduce training time.

3.3.2 Head

The head networks perform classification and regression on features sampled by RoIAlign. The weights of the hidden layers of these networks are also pre-trained for classification, since this yields better results.

In this work, the VGG-16 and ResNet-101 heads comprise all hidden layers after the ones used for the backbone (except for $pool_5$ in VGG-16), with the last classification layer being replaced by the sibling classification and regression layers, that are randomly initialized.

In the code, the networks are loaded with PyTorch and dismantled as a list of layers. The backbone layers are assigned to a backbone variable and the remaining layers, except for the classification layer and $pool_5$, are augmented with the sibling linear layers for classification and regression, and assigned to the head variable. These variables are passed, with a configuration file containing all hyperparameters, to a Faster R-CNN constructor.

3.3.3 RPN

The RPN module is responsible for calculating the classification and regression loss with respect to the sampled anchors (during training), and for calculating the bounding boxes of the regions of interest.

At first, the output feature maps of the backbone are passed to the RPN specific convolutions. The first layer is a 3×3 convolution layer with stride 1, zero padding 1 and 512 filters followed by ReLU, that is used to compute the local features. Those are then passed to sibling 1×1 convolutional layers that compute the $2k$ object scores and $4k$ regression outputs for each position. Note that the final outputs of this process are a $w \times h \times 2k$ score tensor and a $w \times h \times 4k$ regression tensor, where $w \times h$ are the spatial dimensions of the backbone feature maps. Also, the weights of these layers are randomly initialized.

On the PASCAL VOC experiments the anchors are set to bounding boxes of areas 128×128, 256×256 and 512×512, and proportions $1:1$, $1:2$ and $2:1$, to allow the detection of objects of multiple sizes and shapes. This yields $k = 9$ types of anchors. While on the COCO dataset, the RPN is improved, to be able to detect smaller objects, by including anchors of area 64×64 with the same proportions as the previous ones, yielding a total of $k = 12$ types of anchor.

The anchors for each position are then computed. A grid is created containing the center of each anchor and is, then, used to compute a $w \times h \times 4k$ tensor containing, at each spatial position, the 4-tuple coordinates of every anchor at that position. During training, this tensor, together with the score and regression ones, is passed to an RPN loss calculation routine.

Inside the loss calculation function, the anchors, scores and regression tensors are converted, respectively, to $wh \times 4k$, $wh \times 2k$, and $wh \times 4k$ dimension matrices, with each row in each matrix corresponding to the same anchor. Moreover, anchors that cross the image bounds are filtered, following the original implementation [7]. The remaining anchors are then compared to the ground-truth bounding boxes, using the IoU function described above. The resulting IoU matrix is then used to determine the positive and negative anchors using the following criteria: the anchor is positive if it is the one with highest IoU with a ground-truth bounding box, in which case, this ground-truth is its regression target; it is also positive if it has IoU greater than 0.7 with some ground-truth bounding box, and its regression target is the one with which it overlaps the most; otherwise, if the maximum IoU between the anchor and all ground-truth boxes is less than 0.3, it is a negative one. Targets for the regression are the 4-tuples $\mathbf{t} = (t_x, t_y, t_w, t_h)$, computed as in Equations (3.9), (3.10), (3.11) and (3.12), where $\mathbf{G} = (G_x, G_y, G_w, G_h)$ and $\mathbf{P} = (P_x, P_y, P_w, P_h)$ are the ground-truth and proposal bounding box 4-tuples.

$$t_x = \frac{G_x - P_x}{P_w} \tag{3.9}$$

$$t_y = \frac{G_y - P_y}{P_w} \tag{3.10}$$

$$t_w = \log(G_w/P_w) \tag{3.11}$$

$$t_h = \log(G_h/P_h) \tag{3.12}$$

With the targets defined, p positive ($p = 128$ when possible) and $256 - p$ negative anchors are sampled, and the total RPN loss is the average of loss for each anchor, computed as described in Equation (2.26).

Once the loss is computed, the regions of interest bounding boxes are calculated from the anchor and regression tensors, following Equations (2.22), (2.23), (2.24) and (2.25), for each anchor. The regions of interest are then cropped to fit into the image bounds and proposals with width or height smaller than 1 are discarded. These proposals are filtered with a 0.7 threshold NMS and reduced to the top 2000 in train-time and 300 in test-time.

3.3.4 Classification and regression

For training, since only 128 of the 2000 proposals will be sampled to compute the loss, sampling and target computation are done before applying RoIAlign and the head. The IoU matrix between proposals and ground-truth bounding boxes is computed, so as to define class and background targets. For class samples, the proposal IoU must be greater than 0.5 for some ground-truth box, and the target is the ground-truth box with the highest IoU with that proposal, while for background proposals the IoU must be less than 0.1 in PASCAL VOC and 0.0 in COCO. 32 class proposals and 96 background proposals are sampled, and their targets for bounding box regression and classification are calculated. The regression targets are computed as in Equations (3.9), (3.10), (3.11) and (3.12), but they are also normalized to zero mean and unitary variance using the empirical means $\bar{t} = (0, 0, 0, 0)$ and standard deviations $\sigma_t = (0.1, 0.1, 0.2, 0.2)$ (taken from the original implementation repository [7][3], so as to obtain better gradients for optimization. The targets are stored in tensors that will later be used for loss computation.

Once the proposals are defined, their features are passed to the head network via the RoIAlign layer. The head computes the classification and regression outputs, that, in training, are combined with the target tensors to compute the head loss, that is the average of each proposal loss computed as in Equation (2.26). The detection bounding boxes are adjusted using Equations (2.22), (2.23), (2.24) and (2.25), where t is the regression output after reversing the normalization (multiplying by σ_t and adding $\bar{t}$). The bounding boxes are also cropped to the image bounds.

Finally, Faster R-CNN outputs the detections' bounding boxes and scores, and the class-specific and RPN losses.

3.3.5 Postprocessing

For postprocessing, class specific NMS, with a threshold of 0.3, is used to eliminate multiple detections of the same object.

3.4 FPN

The FPN implementation is similar to the Faster R-CNN described above, maintaining most of the hyperparameters. Differences include: using feature maps from multiple levels as inputs in RPN and the head network, image preprocessing to

[3]Official (MATLAB): `https://github.com/ShaoqingRen/faster_rcnn`; Python version (the values are taken from here): `https://github.com/rbgirshick/py-faster-rcnn`; Version that includes Mask R-CNN, RetinaNet, and bells and whistles: `https://github.com/facebookresearch/Detectron` [51].

ensure the pyramid is built correctly, and a different backbone and head for ResNet-101.

3.4.1 Backbone and head

The backbone for ResNet-101 comprises all convolutional layers, excluding only the average pooling and the fully connected classification layer. The head comprises three fully connected layers, two hidden with 1024-D, and the last one for classification and regression.

These networks are implemented by modifying the ResNet code of the PyTorch repository. For the pyramid levels, the outputs of the last residual block of conv2, conv3, conv4 and conv5 are used as the building blocks, and are called, respectively, $\{C_2, C_3, C_4, C_5\}$. A 1×1 convolutional layer, with 256 output channels, is appended to each of the building blocks. Starting from the convolved C_5 feature map, called M_5, the feature maps are upsampled, with $2\times$ nearest neighbor upsampling, to the level directly below and added to the lateral connection, to form the merged feature maps M_i. These M_i maps are then passed to a 3×3 convolutional layer to form the pyramid levels P_i. Another level, P_6, is also included, for larger proposals, by downsampling from P_5 with stride 2. Finally, the backbone returns this $\{P_2, P_3, P_4, P_5, P_6\}$ feature pyramid.

All layers introduced by FPN are linear, since there is no activation function, and are randomly initialized.

Note that, to be able to add the top-down upsampling with the lateral layers, the bottom-up (forward) layers of the network must downsample evenly, that is, all the feature maps of the pyramid (with the exception of the last one) must have an even width and height. To ensure this, the image inputs are preprocessed with zero-padding to have spatial dimensions multiple of 32.

3.4.2 RPN

The RPN-specific layers are applied to each level of the pyramid. These layers are shared among all pyramid levels and return $2k$ object scores and $4k$ regression outputs, with $k = 3$ being the number of different anchor aspect ratios ($2:1$, $1:2$ and $1:1$). Anchor scales are assigned to different levels of the pyramid, so, the feature maps $\{P_2, P_3, P_4, P_5, P_6\}$ are associated, respectively, with anchors of areas 32×32, 64×64, 128×128, 256×256 and 512×512, so that the scales of the proposals generated at each level are proportional to the stride.

The RPN module iterates over the pyramid levels, generating the anchors assigned to each of them and applying the two RPN specific convolutional layers on each level. A difference is that, once the regions of interest are computed from the

regressors and the anchors, only the top 12000 proposals[4] are submitted to NMS. This pre-NMS filter is applied to handle the high number of proposals generated by the lower levels. It also helps reducing inference time. This pre-NMS and the NMS filters are also performed on a per-level basis. Moreover, the proposals are further filtered by a post-NMS filter that returns only the 1000 (2000 for training) proposals with highest score, that also operates on a per-level basis.

After the loop finishes, if it is train time, the RPN loss is calculated, with anchors sampled from the joined set of anchors from all levels, in the same way as in the Faster R-CNN implementation. Once the RPN loss is obtained, the proposals from all levels are concatenated in a single tensor, that is then sorted by score, and only the 1000 (for training, 2000 proposals are used instead) with highest score are returned in this tensor.

3.4.3 Classification and regression

The proposal tensor is then passed to the next stages of the pipeline, that is to have the features extracted by RoIAlign.

Just like in Faster R-CNN, during training, the 128 sampled proposals and their targets are chosen before applying RoIAlign and the head network. Proposal sampling and target assignment follow the same rules as the ones from Section 3.3.

As described in Section 2.3.6, features from regions of interest are extracted from different levels of the feature pyramid. A proposal with dimensions $w \times h$ is assigned to level k defined as in Equation (2.29), with $k_0 = 4$ (i.e., the 224×224 proposals are assigned to the default level of the original Faster R-CNN). Note this implies that proposals of different sizes have similar scales in their assigned feature maps. The implementation of this part is slightly different from the original one, where the pyramid level P_6 is used only for RPN and is disregarded in Fast R-CNN [8]. Here the head network also takes features from the P_6 level.

Once the proposals are assigned to their respective levels, RoIAlign reshapes their features to the head network input shape and they are classified and regressed by the head network. Outputs of the head network, loss calculation and postprocessing follow the same process as described in Section 3.3 for Faster R-CNN.

3.5 Hyperparameters

Table 3.1 presents a summary of hyperparameters over all implementations. "RPN channels" and "RPN kernel size" refer, respectively, to the number of channels and

[4]The original Faster R-CNN implementation uses this pre-NMS filter too, but it was not used in the Faster R-CNN implementation from this work.

the kernel size of the first additional convolutional layer of the RPN, "RPN NMS threshold" is the threshold for filtering the proposals generated by the RPN with NMS, "RoI size" is the number of the spatial dimensions output by RoIAlign (note that changing this hyperparameter requires changing the head network), "anchor areas" is the set of areas that determine the scales of anchors in the RPN and "anchor ratios" is the set of aspect ratios for anchors of all sizes.

Other than the ones aforementioned, FPN has hyperparameters in the number of filters in the introduced layers, which is 256 in every one of those layers in this work, and in the number of regions of interest filtered before the per-layer NMS in RPN, which is 12000 here.

Table 3.1: Summary of hyperparameters used in the implementations of Faster R-CNN and FPN. Some hyperparameters are fixed across all implementations: 512 RPN channels, 3 RPN kernel size, 0.7 RPN NMS threshold, $\{(1:1), (1:2), (2:1)\}$ anchor ratios.

Hyperparameter	**System**		
	Faster R-CNN		FPN
	VGG-16	ResNet-101	ResNet-101
Anchor areas	$\{128^2, 256^2, 512^2\}$	$\{128^2, 256^2, 512^2\}$	$\{32^2, 64^2, 128^2, 256^2, 512^2\}$
Test proposals	300	300	1000
RoI size	7	14	7

Chapter 4

Results

This chapter comprehends the datasets specifications (in Section 4.1), details on preprocessing (in Section 4.2), description of the experiments and results on the datasets Microsoft COCO (in Section 4.4) and PASCAL VOC (in Section 4.5), discussion of the inference time of the implementations developed (in Section 4.6), and a discussion of the results (in Section 4.7). The results obtained in the original Faster R-CNN and FPN are also presented for comparison. Section 4.9 discusses modifications made on the implementation that reduced the gap, present on the results from the previous sections, with the baseline implementations. Section 4.9 also presents improved results with the newly trained models.

The models evaluated were produced using the code available at the official repository[1]. The results were generated using the official codes from each challenge[2].

4.1 Datasets

Training and testing are done on the COCO and PASCAL VOC datasets. In both cases the dataset provides multiple images with corresponding annotations. An annotation contains information about the bounding box and the classification of every object in that image that belongs to one of the classes of interest of that dataset. The dataset may contain information for other tasks, such as segmentation, but that information is not used in this work.

4.1.1 COCO

The COCO [24] 2017 dataset comprehends 123k labeled images, with a 118k/5k training and validation split. There is a set of 80 classes for detection, which includes

[1] `https://gitlab.com/pedrocayres/faster_rcnn_pytorch`. This repository may be updated after the publication of this work and, therefore, may contain further improvements.

[2] See `http://host.robots.ox.ac.uk/pascal/VOC/` and `http://cocodataset.org` for more details.

the classes annotated in PASCAL VOC.

The COCO detection challenge compares methods based on mAP@[.5, .95] evaluation. This metric is calculated by taking the average over mAP computed at each IoU threshold from 0.5 to 0.95, with a 0.05 step. Detections are limited to 100 per image.

4.1.2 PASCAL VOC

Two complementary datasets from PASCAL VOC [25, 26] are employed, the 2007 and 2012 sets. The PASCAL VOC 2007 contains 9963 images, with a 5k/5k split for trainval (union of training and validation) and test, while the PASCAL VOC 2012 comprehends 11540 images for trainval. All these datasets comprise 20 classes for detection. In this work all experiments are tested in the PASCAL VOC 2007 test set, while the trainval sets are used to train the detectors.

The PASCAL VOC 2007 detection challenge uses the mAP@0.5 to evaluate the method performances, but it is slightly different from the one described in Section 2.3. The PASCAL VOC 2007's AP is computed by taking the average precision from 11 sample points of the interpolated precision/recall curve (a range of values from 0 to 1 with a step of 0.1).

4.2 Preprocessing

As is common practice in CNN, the input images were normalized to have unitary variance and zero mean on each channel, with the mean and variance values taken from the ImageNet dataset.

For both training and testing the images are re-scaled such that its smallest side have a length of 600 pixels. If this scale would lead the image to have more than 1000 pixels on its largest side then, instead, the image is re-scaled such that the largest side measures 1000 pixels. Higher resolutions can be used, but will consume more memory and take longer for inference.

Also, during training, in order to improve generalization, images are horizontally flipped with a 50% chance.

4.3 Hyperparameters for training

The hyperparameters used only for training the Faster R-CNN and FPN are specified as: 256 “RPN batch size” is the number of anchors per mini-batch, 0.7 “RPN positive IoU” is the lower bound for the IoU overlap an anchor needs to have with a ground-truth to be considered a positive object sample, 0.3 “RPN negative IoU” is the

upper bound for the IoU an anchor can have with a ground-truth to be considered a negative object sample, 128 “class batch size” is the number of regions of interest per mini-batch. The ratio for object over background samples in RPN is 128 : 128 (when there are enough positive object samples) and the ratio for class over background region of interest samples in the final classifier loss is 32 : 96.

Optimization hyperparameters vary and are described in the experiment sections (Sections 4.4 and 4.5).

4.4 Experiments on COCO

The three implementations presented in Chapter 3, that is Faster R-CNN with VGG-16 and ResNet-101, and FPN with ResNet-101, are trained on the COCO 2017 train set and tested on the COCO 2017 validation set.

Following the original work, some implementation details are changed in Faster R-CNN for these experiments. Anchors of area 64^2, with the same aspect ratios as defined before, are included so that the network can detect smaller objects better. Moreover, the lower bound IoU overlap for the background region of interest samples is changed from 0.1 to 0.0.

4.4.1 Optimization

All experiments optimize the RPN and the network head concurrently using the approximate joint training approach. Mini-batches consists of 256 anchors and 128 regions of interest taken from a single image. For the COCO experiments, the weight updating rule is given by SGD with learning rate 0.001 and momentum 0.9 for 6 epochs and learning rate 0.0001 for the next 6 epochs. For regularization, aside from the dropout and batch normalization layers, a weight decay of 0.0005 is used.

4.4.2 Results

The results obtained by training the Faster R-CNN, with ResNet-101 and VGG-16, and FPN, with ResNet-101, on the COCO 2017 dataset are shown in Table 4.1. Here AP is mAP@[.5, .95] and AP_s, AP_m, AP_l are the mAP values for small, medium and large size objects, respectively, following the definitions in the COCO paper [24].

The Faster R-CNN with ResNet-101 implementation performed worse than the original by 1.6% in mAP@[.5, .95], the FPN implementation also performed worse, by 9.7% mAP@[.5, .95], but this can be partially explained by the fact that it was trained with 600 pixels on the shorter side, whereas the original used 800 pixels instead. For VGG, there is a difference of 0.2% mAP@[.5, .95] in comparison with the original.

Observe that, as was expected, FPN improves the small objects detection (AP_s) in comparison with Faster R-CNN. Indeed, Faster R-CNN with ResNet-101 achieves better AP_l and AP_m, but, nevertheless, loses to FPN in mAP@[.5, .95] and mAP@0.5, since FPN is significantly better in AP_s.

Table 4.1: Detection results on the COCO validation set. Training was done on the COCO training set. The size column is the size of the smallest side of the image input of that model, while the RoIOp. column determines the donwsampling operation performed on the regions of interest of that model.

Method	**Backbone**	**Size**	**RoIOp.**	**AP@0.5**	**AP**	**AP_s**	**AP_m**	**AP_l**
Faster R-CNN (ours)	VGG-16	600	RoIAlign	37.7	21.0	7.5	23.2	30.6
Faster R-CNN† [7]	VGG-16	600	RoIPool	41.5	21.2	-	-	-
Faster R-CNN (ours)	ResNet-101	600	RoIAlign	42.0	25.6	8.7	28.9	39.5
Faster R-CNN† [22]	ResNet-101	600	RoIPool	48.4	27.2	-	-	-
FPN (ours)	ResNet-101	600	RoIAlign	43.7	26.0	10.9	28.6	38.0
FPN [8]	ResNet-101	800	RoIPool	-	35.7	-	-	-

† The results from the baseline Faster R-CNN models were obtained with an older version of the COCO dataset, from 2015, when the usual split was 83k images for training and 41k images for validation, and this split differs from the 118k/5k for train/minival split, described in Section 4.1, that is suggested since 2017.

4.5 Experiments on PASCAL VOC

Two experiments are realized in the PASCAL VOC datasets. In the first one, all three implementations are trained on the VOC 2007 trainval set (union of the training and validation sets) and evaluated on the VOC 2007 test set, while in the second one, the same implementations are trained on the VOC 2007+2012 trainval set (union of VOC 2007 and VOC 2012 trainval, about 16k images) and tested on the VOC 2007 test set.

4.5.1 Optimization

On the VOC 2007+2012 experiment, the optimization algorithm is SGD with learning rate 0.001 and momentum 0.9 for 6 epochs and then learning rate 0.0001 for the next 6 epochs. The training on VOC 2007 also uses SGD with learning rate 0.001 and momentum 0.9, but the learning rate decreases after 12 epochs, continuing the next 6 epochs with a step of 0.0001. For regularization, aside from the dropout and batch normalization layers, a weight decay of 0.0005 is used.

4.5.2 Results

The results obtained by training the Faster R-CNN, with ResNet-101 and VGG-16, and FPN, with ResNet-101, on the PASCAL VOC 2007 dataset are shown on Table 4.2. Here, this work's VGG-16 falls short by 6.8% mAP when compared to the original.

Table 4.2: Detection results on the PASCAL VOC 2007 test set. Training was done on the PASCAL VOC 2007 trainval set.

Method	**Backbone**	**RoIOp.**	**mAP**
Faster R-CNN (ours)	VGG-16	RoIAlign	63.1
Faster R-CNN [7]	VGG-16	RoIPool	69.9
Faster R-CNN (ours)	ResNet-101	RoIAlign	69.0
FPN (ours)	ResNet-101	RoIAlign	71.8

The models were also trained on a dataset formed by the union of the VOC 2007 and VOC 2012 datasets, with inference being made on VOC 2007 test set, as displayed in Table 4.3. Since there is more training data in comparison with the first experiment, the mAPs of the Faster R-CNN and FPN with ResNet-101 implementations were raised by 3.3% and 2.7%, respectively, while Faster R-CNN with VGG-16 improved by 5% mAP. Here, the result for Faster R-CNN with ResNet-101 is worse by 4.1% in comparison with the original, while the difference for Faster R-CNN with VGG-16 is 5.1%.

Table 4.3: Detection results on the PASCAL VOC 2007 test set. Training was done on the union of PASCAL VOC 2007 trainval and PASCAL VOC 2012 trainval sets.

Method	**Backbone**	**RoIOp.**	**mAP**
Faster R-CNN (ours)	VGG-16	RoIAlign	68.1
Faster R-CNN [7]	VGG-16	RoIPool	73.2
Faster R-CNN (ours)	ResNet-101	RoIAlign	72.3
Faster R-CNN [22]	ResNet-101	RoIPool	76.4
FPN (ours)	ResNet-101	RoIAlign	74.5

4.6 Time profiling

The time profiling for the three networks implemented in this work can be seen in Table 4.4. Note that, for all systems, there is a bottleneck in the NMS operation on the proposals that come from RPN. The original Faster R-CNN paper reports, for a Faster R-CNN with VGG-16 tested on a K40 GPU, 198 ms total inference time: 141 ms from the backbone, 10 ms from the additional RPN layers and 47 ms for

NMS, RoIPool, fully connected, and softmax layers [7]. In comparison, the time for NMS in this work is 264 ms for VGG-16, which is more than 5 times greater than 47 ms, even when disregarding the time from the other layers, whereas the layers from backbone and head are around 5 times faster than the original.

These results also show that FPN can achieve higher precision while also being faster. This derives mostly from the much lighter head design that takes only 4 ms compared to Faster R-CNN's 67 ms, while the more complex backbone only adds 7 ms. NMS is faster in FPN only because of the pre-NMS filter that takes only the 12000 best proposals.

Table 4.4: Time spent in each module of the networks for one detection during test time. RPN time refer to the RPN-specific layers and calculating the proposals, and NMS time is the time spent on NMS in the proposals from RPN only. Other includes RoIAlign, softmax and class specif NMS. These values are average values computed over 1000 samples. All tests were done on a Nvidia GTX 1080 Ti GPU and an Intel Core i7-6850K CPU.

System		**Time (ms)**					
Method	**Backbone**	**Backbone**	**RPN**	**NMS**	**Head**	**Other**	**Total**
Faster R-CNN	VGG-16	27	2	264	9	17	320
Faster R-CNN	ResNet-101	27	3	278	67	28	404
FPN	ResNet-101	34	15	206	4	21	281

4.7 Discussion

Among the implementations developed in this work, the performance of the FPN is superior on all experiments done on COCO and PASCAL VOC, corroborating the observations made in the original FPN article. All of them are still worse than the original Faster R-CNN with ResNet-101, possibly because of differences on hyperparameters assignments and training schedules.

The best performance on the COCO experiment, among all systems presented here, is achieved by the original FPN with ResNet-101, but this can be attributed to the fact that it infers on a larger resolution than the other methods.

It is important to note that the training algorithm presented here differs from the original one in some aspects. The results from the papers use the 4-step alternate training with the following rules:

- Faster R-CNN [7]: 0.001 learning rate SGD for 60k mini-batches (256 anchors sampled from one image) and then 0.0001 for 20k mini-batches, to train RPN. Fast R-CNN is trained with 0.001 learning rate SGD for 30k mini-batches (128 RoIs from 2 images) and then with 0.0001 for 10k mini-batches on VOC 2007,

and 60k mini-batches in total on VOC 2007+2012. For COCO the training is done with 0.003 SGD for 240k mini-batches (in that case, because training is done in parallel on 8 GPUs, a mini-batch comprises 8 images for RPN and 16 for Fast R-CNN) and then for $80k$ mini-batches with a learning rate of 0.0003, for both RPN and Fast R-CNN.

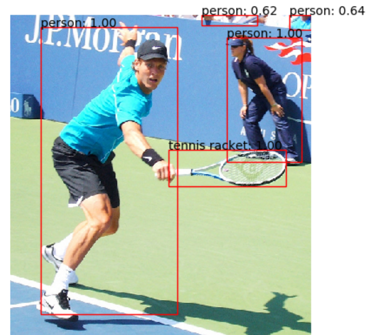
- Faster R-CNN with FPN [8]: RPN training uses 0.02 learning rate SGD for 30k mini-batches (16 images with 256 anchors per image) and then uses 0.002 learning rate for the next 10k mini-batches. Fast R-CNN uses 0.02 learning rate SGD for 120k mini-batches (16 images with 512 RoIs per image) and then uses 0.002 learning rate for 40k mini-batches. These values are for training on COCO and with images resized such that their shortest side has 800 pixels, instead of 600 pixels as in the other experiments.

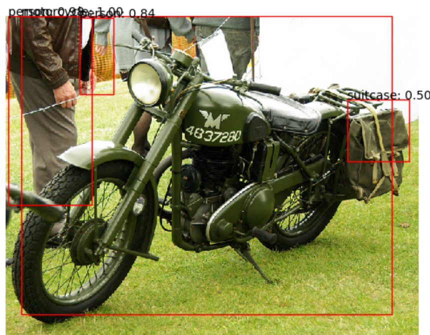
The 4-step alternate training involves: (1) training an RPN from a pre-trained backbone; (2) training a separate Fast R-CNN from the same pre-trained backbone using proposals generated from this RPN; (3) initializing the RPN with the weights from the trained backbone of Fast R-CNN, fixing the backbone and fine-tuning the specific RPN layers; (4) fine-tuning the head of the Fast R-CNN, fixing all other layers. On the other hand, in the training method implemented in this work, RPN and Fast R-CNN are trained together, using the same mini-batches consisting of 256 anchors and 64 RoIs from one image, with the following training rules, for both FPN and Faster R-CNN: 0.001 learning rate SGD for 60k mini-batches and then 0.0001 for 30k mini-batches on VOC 2007. VOC 2007+2012 uses the same learning rates in the same order, using 60k mini-batches for each learning rate. COCO training is done with a learning rate of 0.001 for 704k mini-batches and 0.0001 for the next 704k mini-batches. While the number of mini-batch iterations may be greater than those used in the original for each singular step, the mini-batches are much smaller in COCO and FPN. Moreover, the head network and the RPN layers are trained for more iterations during 4-step alternate training.

4.8 Examples

Random samples of detections made on the COCO minival set by FPN on ResNet-101 can be seen in Figure 4.1.

Figures 4.2 and 4.3 show examples of object detection carried out on day-time / night-time street crossing videos. Note that the network is working outside of its domain, since the frames of these videos have a significantly lower quality than the images in the COCO dataset. Nevertheless, the detector successfully detects multiple car instances, seemingly achieving good precision in the day-time video.





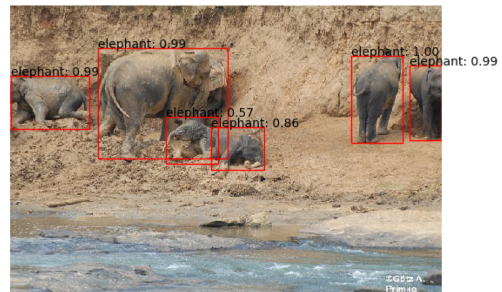


Figure 4.1: Random samples of COCO minival detected by the trained FPN (with a threshold of 0.5). In the top-left image, some laptops were correctly identified, while some were confused with chairs, and one of the detected laptops has a misplaced bounding box. There were some keyboard detections, but laptop keyboards are not considered keyboards in the ground-truth. The backpack was mistook for a handbag / suitcase. The tangled wires were detected as a bicycle and part of the router was mistaken for a book. In the top-right image, the persons and the racket were detected correctly, the detector even found people in the stands. In the bottom-left image, one person and the motorcycle were identified correctly, and the handbag was incorrectly classified. In the bottom-right image, all elephants were identified correctly.

Detection seems to be worse in the night-time video, except for the traffic light that is consistently detected.

4.9 Improvements

In order to reduce the accuracy gap between the systems trained in this work and the baselines, a series of improvements were implemented, following the baseline repositories of Faster R-CNN[3] and Mask R-CNN[4]. These changes affect the pre-NMS filters, region of interest mini-batch sampling, loss function, inference and optimization. Further experiments are executed and the results are presented in this section. There was also a bug in the FPN code (all regions of interest were being projected in the lowest level) that was fixed.

4.9.1 Pre-NMS filters

Pre-NMS filters are introduced in the RPNs of all models. They are simple filters that keep only the top scoring bounding boxes regressed from the anchors. For all pure Faster R-CNN models the number of bounding boxes kept pre-nms are 12000 during training and 6000 during inference. For the FPN models these numbers are 2000 per level for training and 1000 per level for inference (this was already implemented, but the numbers were corrected using values from the baseline implementation repository). This reduced the RPN NMS time to around 36 ms on Faster R-CNN and 20 ms during test.

4.9.2 Mini-batch sampling

The ratio of regions of interest sampled to train the head network are 25% for class and 75% for background, from the total mini-batch defined size (128 for the experiments described in the previous sections). Previously the class and background regions of interest were sampled independently, so, if the RPN did not output enough class (or background) regions to attend the defined ratio, the mini-batch could be smaller than the defined size.

The improved methods sample using a sampling strategy similar to the one used for sampling training anchors, that is, if there are not enough class samples to fulfill the 25% ratio, the mini-batch is completed with background samples. Moreover, during training, the ground-truth bounding boxes are appended to the regions of interest bounding boxes generated by the RPN, which means they can be selected

[3]`https://github.com/rbgirshick/py-faster-rcnn`

[4]`https://github.com/facebookresearch/Detectron`

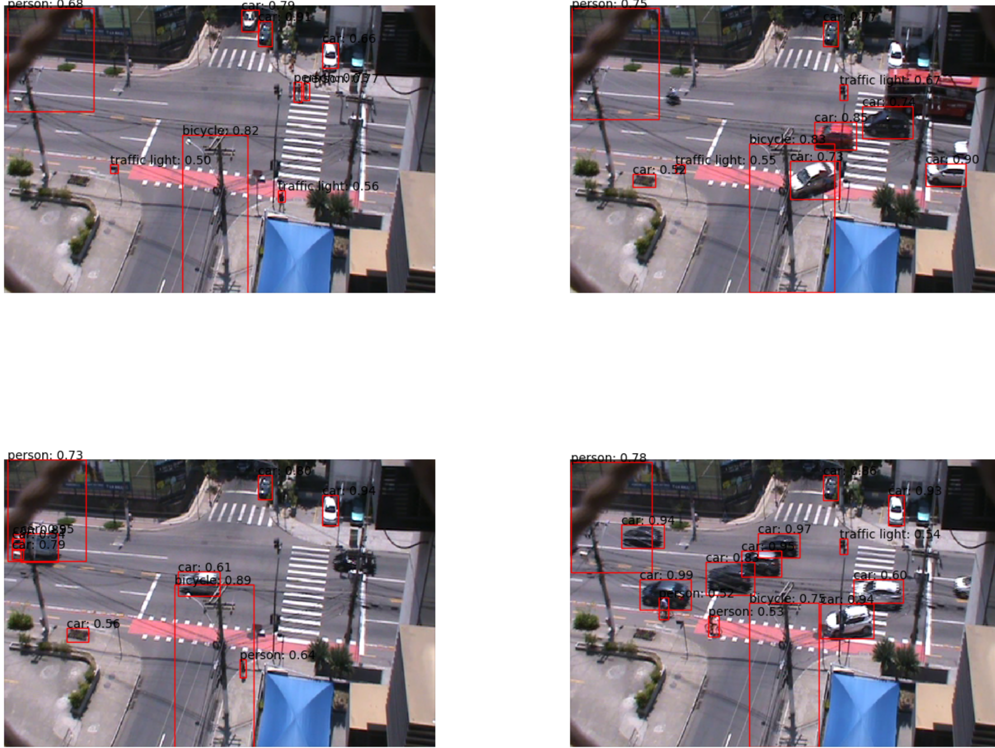


Figure 4.2: Detections made by the FPN on the frames of a video of a street crossing. Only detections of persons, bicycles, cars, motorcycles, buses, trucks, traffic lights and stop signs, that have a confidence greater than 0.5, are displayed. The detector consistently detects a bicycle in the middle pole, and a person in the top-right of the images near the safety net. In the top-left image, it detected some cars and people correctly, but it misclassified a person and a street sign as traffic lights. In the top-right image, the detector identified all cars correctly, albeit with one false positive. It also detected the traffic light in the middle of the image. In the bottom-left image, the truck was classified as a car and generated three false detections. Moreover, a person in the sidewalk was also detected. In the bottom-right image, almost all cars were detected. It also detected the traffic light and people riding bicycles, but missed the bicycles themselves.

Figure 4.3: Detections made by the FPN on the frames of a video of a street crossing at night. In all images the detector finds the traffic light, but incorrectly detects persons in the right and left side of the image. In the top-left image, four of the five cars are detected, although regression seems worse than in the daytime video. The top-right image presents two incorrect car detections. One of them is a false positive generated by a motorcycle. In the bottom-left image, there is one more incorrect person detection (at the car headlight), but the detector correctly identified the pedestrian traffic light in the top. In the bottom-right image, four out of six cars are detected.

as a class sample in the mini-batch. Note that this ensures that all images that contain annotated objects produce class samples in the mini-batch.

Moreover, like in the FPN paper [8], the FPN also considers the anchors that cross the image bounds when sampling. Previously it ignored those samples during training, like in the Faster R-CNN paper [7].

4.9.3 Loss function

The smooth L_1 function, described in Equation 2.28, can be generalized as a function $s_\sigma : \mathbb{R} \to \mathbb{R}$ with a parameter $\sigma \in \mathbb{R}^+$, defined as in Equation 4.1 (note that all such functions are still differentiable). Likewise, the regression loss function, defined in Equation 2.27, can be redefined as in Equation 4.2, where c' is the ground-truth class, $\mathbf{t}^{c'}$ is the 4-tuple bounding box coordinates calculated by the network on the ground-truth class and $\mathbf{t}$ is the 4-tuple of the ground-truth bounding box. Note that smooth L_1 and the original loss function are recovered when $\sigma = 1$.

$$s_\sigma(x) = \begin{cases} 0.5x^2\sigma^2 & \text{if } |x| < \frac{1}{\sigma^2} \\ |x| - \frac{0.5}{\sigma^2} & \text{otherwise} \end{cases} \tag{4.1}$$

$$L_{reg}(\mathbf{t}^{c'}, \mathbf{t}', c') = \sum_{i \in \{x,y,w,h\}} s_\sigma(t_i^{c'} - t'_i) \tag{4.2}$$

Using these definitions, the loss function of the RPN bounding box regression is modified to use $\sigma = 3$ (the one in the previous sections is equivalent to set $\sigma = 1$). Bounding box regression loss in the head network remains unchanged.

4.9.4 Inference

At test time, to determine the detected bounding boxes, the bounding boxes proposed by the RPN are regressed for every possible class. Low scoring detections are filtered by applying a 0.05 threshold on the outputs of the softmax classification layer. A per class NMS with 0.3 overlap threshold is applied and the detections from all classes are joined and ordered by score. Then, the inference process returns the 100 detections with the highest scores.

This differs from the previously implemented method, where a region proposed by the RPN was regressed only to the class with the highest score with respect to that region. So, for a task consisting of C categories and before the filters are applied, the new method proposes C detections per region of interest, whereas the previous method proposed at most 1 detection per region of interest (0 when background had the highest score).

4.9.5 Optimization

Following practices of the baseline repositories, the optimization algorithm is modified, changing the behaviour of weight decay and learning rate.

The weight decay for methods that employ ResNet-101 are changed to 0.0001, as opposed to 0.0005, the same value that was used to optimize for the classification task and obtain the pre-trained weights. Moreover weight decay is not applied to the biases anymore, in any method. Another change is that, for VGG-16, the learning rate is doubled for the biases.

4.9.6 Experiments on COCO

More experiments are performed, with the above modifications. Faster R-CNN with VGG-16 and ResNet-101, and FPN with ResNet-101 are trained on the COCO 2017 train set and tested on the COCO 2017 validation set and the COCO 2017 test-dev set. FPN with ResNet-101 is trained for two separate models, one where the smallest input image side is resized to 600 pixels (and largest side at most 1000 pixels) and another for where the smallest input image side is resized to 800 (and largest side at most 1333 pixels).

The models are optimized like in Section 4.4, but with the corrections described in Section 4.9.5. The new model of the FPN that operates on 800 pixels images is trained with a larger mini-batch with 512 regions of interest and use a per class NMS of 0.5 (instead of 0.3, as in the models that are trained on 600 pixels images).

Results

With the modifications, the models achieved better results in the COCO validation set, as can be seen by comparing Tables 4.1 and 4.5. The methods implemented in this work show better mAP@[.5, .95] performance when compared to their baseline counterparts, which is expected, since the original implementations use RoIPool (RoIAlign should improve performance by around 1.1% mAP@[.5, .95], according to the Mask R-CNN article [8]). While mAP@0.5 is also higher on the Faster R-CNN with ResNet-101 model implemented here, when compared to its baseline counterpart, the Faster R-CNN with VGG-16 model trained here is 0.3% mAP@0.5 worse than the original.

Results from evaluating the models on the COCO test-dev set can be seen in Table 4.6. Faster R-CNN with ResNet-101 performs only 1.0% worse than the more heavily engineered Faster R-CNN+++ with ResNet-101, that uses image pyramids on inference. Faster R-CNN with VGG-16 still performs worse by 1.3% mAP@0.5 compared to the original, but is better in mAP@[.5, .95] by 0.5%. The FPN model with large inputs performs better than both the published RoIPool and RoIAlign

versions, it is 0.8% mAP@[.5, .95] and 2.0% mAP@0.5 higher than the Faster R-CNN with ResNet-101-FPN version from the Mask R-CNN paper [8].

The FPN with ResNet-101 models trained here are the best models, among all presented models presented here, for their respective size inputs. The larger image scales improve 2.7% mAP@[.5, .95] and 3.7% mAP@0.5, which comes mostly from a better AP on smaller scale objects (as can be seen in Tables 4.5 and 4.6, the differences are larger in AP_s and smaller in AP_l).

Table 4.5: Detection results on the COCO validation set with the updated methods. Training was done on the COCO training set.

Method	**Backbone**	**Size**	**RoIOp.**	**AP@0.5**	**AP**	**AP_s**	**AP_m**	**AP_l**
Faster R-CNN (ours)	VGG-16	600	RoIAlign	41.2	22.4	9.2	25.2	31.3
Faster R-CNN† [7]	VGG-16	600	RoIPool	41.5	21.2	-	-	-
Faster R-CNN (ours)	ResNet-101	600	RoIAlign	53.7	33.6	15.1	37.8	49.0
Faster R-CNN† [22]	ResNet-101	600	RoIPool	48.4	27.2	-	-	-
FPN (ours)	ResNet-101	600	RoIAlign	56.7	35.2	17.8	39.2	48.8
FPN (ours)	ResNet-101	800	RoIAlign	60.0	37.7	22.1	41.9	49.0
FPN [8]	ResNet-101	800	RoIPool	-	35.7	-	-	-

† The results from the baseline Faster R-CNN models were obtained with an older version of the COCO dataset, from 2015, when the usual split was 83k images for training and 41k images for validation, and this split differs from the 118k/5k for train/minival split, described in Section 4.1, that is suggested since 2017.

Table 4.6: Detection results on the COCO test-dev set with the updated methods.

Method	**Backbone**	**Size**	**RoIOp.**	**AP@0.5**	**AP**	**AP_s**	**AP_m**	**AP_l**
Faster R-CNN (ours)	VGG-16	600	RoIAlign	41.4	22.4	9.7	24.6	30.1
Faster R-CNN [7]	VGG-16	600	RoIPool	42.7	21.9	-	-	-
Faster R-CNN (ours)	ResNet-101	600	RoIAlign	54.3	33.9	14.9	36.9	47.9
Faster R-CNN+++ [22]	ResNet-101	†	RoIPool	55.7	34.9	15.6	38.7	50.9
FPN (ours)	ResNet-101	600	RoIAlign	57.4	35.4	17.9	38.3	47.3
FPN (ours)	ResNet-101	800	RoIAlign	61.1	38.1	21.7	41.3	47.6
FPN [8]	ResNet-101	800	RoIPool	59.1	36.2	18.2	39.0	48.2
FPN [3]	ResNet-101	800	RoIAlign	59.6	37.3	19.8	40.2	48.8

† Faster R-CNN+++ uses box refinement, global context and multi-scale testing to improve the competition results.

4.9.7 Experiments on PASCAL VOC

The experiments from Section 4.5 are also performed with the aforementioned modifications. The updated results are provided below.

Results

Tables 4.7 and 4.8 show the results of the experiments evaluated on the PASCAL VOC 2007 test set. The Faster R-CNN with VGG-16 models implemented here did not reach the baseline mAP, being 3.1% mAP worse while training on VOC 2007 trainval and 2.4% mAP worse while training on VOC 2007+2012. On the VOC 2007+2012 experiment, the Faster R-CNN with ResNet-101 model seems to have equivalent performance (there is a difference of 0.2%) when compared with the original implementation, when it should be a little better because of RoIAlign.

On both experiments, the model with the best performance was the FPN model trained in this work.

Table 4.7: Detection results on the PASCAL VOC 2007 test set with the updated methods. Training was done on the PASCAL VOC 2007 trainval set.

Method	**Backbone**	**RoIOp.**	**mAP**
Faster R-CNN (ours)	VGG-16	RoIAlign	65.8
Faster R-CNN [7]	VGG-16	RoIPool	69.9
Faster R-CNN (ours)	ResNet-101	RoIAlign	72.5
FPN (ours)	ResNet-101	RoIAlign	73.7

Table 4.8: Detection results on the PASCAL VOC 2007 test set with the updated methods. Training was done on the union of PASCAL VOC 2007 trainval and PASCAL VOC 2012 trainval sets.

Method	**Backbone**	**RoIOp.**	**mAP**
Faster R-CNN (ours)	VGG-16	RoIAlign	70.8
Faster R-CNN [7]	VGG-16	RoIPool	73.2
Faster R-CNN (ours)	ResNet-101	RoIAlign	76.2
Faster R-CNN [22]	ResNet-101	RoIPool	76.4
FPN (ours)	ResNet-101	RoIAlign	76.9

Chapter 5

Conclusion

This work presented implementations for Faster R-CNN and FPN using the VGG-16 and ResNet-101 networks as the backbones. The produced network architectures are able to detect the specified objects, as noted in the examples.

For comparison with the original implementations, performance was measured using the mAP on the Microsoft COCO 2017 and PASCAL VOC 2007 test sets. The results of the experiments show worse performance on the PASCAL VOC experiments, when compared to the equivalent baselines, and similar performance on the COCO experiments, where the FPN model trained in this work is shown to be slightly better than the baseline. It must be considered that not all hyperparameters are equalized between the implementations, moreover, slight changes in the implementations may also contribute to the discrepancies found in the results. For example, it is possible that using a longer training schedule on the implementation provided here would be enough to equalize these results.

5.1 Future Work

Future work includes improvements both in accuracy and inference time of the implementations. Work can also be done to implement other modules and to apply the architecture to different tasks, such as instance segmentation [3], dense captioning [2] and autonomous driving [1].

On the improvements end, CUDA C/C++ implementations of NMS and IoU could help to reduce the time spent on filtering proposals in RPN. The implementation also lacks a routine to feedforward multiple images in parallel, which would be useful for reducing training time and would help in comparisons with the original work [7, 8], since some of their experiments were done with multiple images and GPUs. Other improvements are to equalize the number of training iterations performed here and in the original experiments for a better comparison.

Another line of work is to implement tasks such as instance segmentation, that

can be done by including an FCN [23] in the head network and following the steps of Mask R-CNN [3]. DenseCap [2] is based on Faster R-CNN and could be developed from this framework. It uses this architecture combined with a LSTM (long short-term memory) [52] network to solve the dense captioning task, that involves simultaneously localizing regions in an image and describing them.

Bibliography

[1] CHEN, X., MA, H., WAN, J., et al. "Multi-view 3D Object Detection Network for Autonomous Driving". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6526–6534, July 2017. doi: 10.1109/CVPR.2017.691.

[2] JOHNSON, J., KARPATHY, A., FEI-FEI, L. "DenseCap: Fully Convolutional Localization Networks for Dense Captioning". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4565–4574, June 2016. doi: 10.1109/CVPR.2016.494.

[3] HE, K., GKIOXARI, G., DOLLÁR, P., et al. "Mask R-CNN". In: *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, Oct 2017. doi: 10.1109/ICCV.2017.322.

[4] GIRSHICK, R., DONAHUE, J., DARRELL, T., et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, June 2014. doi: 10.1109/CVPR.2014.81.

[5] HE, K., ZHANG, X., REN, S., et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 37, n. 9, pp. 1904–1916, Sep. 2015. ISSN: 0162-8828. doi: 10.1109/TPAMI.2015.2389824.

[6] GIRSHICK, R. "Fast R-CNN". In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, Dec 2015. doi: 10.1109/ICCV.2015.169.

[7] REN, S., HE, K., GIRSHICK, R., et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 39, n. 6, pp. 1137–1149, June 2017. ISSN: 0162-8828. doi: 10.1109/TPAMI.2016.2577031.

[8] LIN, T., DOLLÁR, P., GIRSHICK, R., et al. "Feature Pyramid Networks for Object Detection". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 936–944, July 2017. doi: 10.1109/CVPR.2017.106.

[9] LIU, W., ANGUELOV, D., ERHAN, D., et al. "SSD: Single Shot MultiBox Detector". In: Leibe, B., Matas, J., Sebe, N., et al. (Eds.), *Computer Vision – ECCV 2016*, pp. 21–37, Cham, 2016. Springer International Publishing. ISBN: 978-3-319-46448-0. doi: 10.1007/978-3-319-46448-0_2.

[10] FU, C.-Y., LIU, W., RANGA, A., et al. "DSSD : Deconvolutional Single Shot Detector", 2017. Disponível em: <http://arxiv.org/abs/1701.06659>.

[11] REDMON, J., DIVVALA, S., GIRSHICK, R., et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, June 2016. doi: 10.1109/CVPR.2016.91.

[12] REDMON, J., FARHADI, A. "YOLO9000: Better, Faster, Stronger". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, July 2017. doi: 10.1109/CVPR.2017.690.

[13] DAI, J., LI, Y., HE, K., et al. "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: Lee, D. D., Sugiyama, M., Luxburg, U. V., et al. (Eds.), *Advances in Neural Information Processing Systems 29*, Curran Associates, Inc., pp. 379–387, 2016. Disponível em: <http://papers.nips.cc/paper/6465-r-fcn-object-detection-via-region-based-fully-convolutional-networks.pdf>.

[14] LIN, T., GOYAL, P., GIRSHICK, R., et al. "Focal Loss for Dense Object Detection". In: *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2999–3007, Oct 2017. doi: 10.1109/ICCV.2017.324.

[15] LOWE, D. G. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*, v. 2, pp. 1150–1157 vol.2, Sep. 1999. doi: 10.1109/ICCV.1999.790410.

[16] LOWE, D. G. "Distinctive Image Features from Scale-Invariant Keypoints", *International Journal of Computer Vision*, v. 60, n. 2, pp. 91–110, Nov 2004.

ISSN: 1573-1405. doi: 10.1023/B:VISI.0000029664.99615.94. Disponível em: <https://doi.org/10.1023/B:VISI.0000029664.99615.94>.

[17] DALAL, N., TRIGGS, B. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, v. 1, pp. 886–893 vol. 1, June 2005. doi: 10.1109/CVPR.2005.177.

[18] FELZENSZWALB, P. F., GIRSHICK, R. B., MCALLESTER, D., et al. "Object Detection with Discriminatively Trained Part-Based Models", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 32, n. 9, pp. 1627–1645, Sep. 2010. ISSN: 0162-8828. doi: 10.1109/TPAMI.2009.167.

[19] VIOLA, P., JONES, M. J. "Robust Real-Time Face Detection", *International Journal of Computer Vision*, v. 57, n. 2, pp. 137–154, May 2004. ISSN: 1573-1405. doi: 10.1023/B:VISI.0000013087.49260.fb. Disponível em: <https://doi.org/10.1023/B:VISI.0000013087.49260.fb>.

[20] PAPAGEORGIOU, C. P., OREN, M., POGGIO, T. "A general framework for object detection". In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pp. 555–562, Jan 1998. doi: 10.1109/ICCV.1998.710772.

[21] SIMONYAN, K., ZISSERMAN, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*, 2015.

[22] HE, K., ZHANG, X., REN, S., et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016. doi: 10.1109/CVPR.2016.90.

[23] LONG, J., SHELHAMER, E., DARRELL, T. "Fully convolutional networks for semantic segmentation". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, June 2015. doi: 10.1109/CVPR.2015.7298965.

[24] LIN, T.-Y., MAIRE, M., BELONGIE, S., et al. "Microsoft COCO: Common Objects in Context". In: Fleet, D., Pajdla, T., Schiele, B., et al. (Eds.), *Computer Vision – ECCV 2014*, pp. 740–755, Cham, 2014. Springer International Publishing. ISBN: 978-3-319-10602-1. doi: 10.1007/978-3-319-10602-1_48.

[25] EVERINGHAM, M., VAN GOOL, L., WILLIAMS, C. K. I., et al. "The Pascal Visual Object Classes (VOC) Challenge", *International Journal of Computer Vision*, v. 88, n. 2, pp. 303–338, Jun 2010. ISSN: 1573-1405. doi: 10.1007/s11263-009-0275-4. Disponível em: <https://doi.org/10.1007/s11263-009-0275-4>.

[26] EVERINGHAM, M., ESLAMI, S. M. A., VAN GOOL, L., et al. "The Pascal Visual Object Classes Challenge: A Retrospective", *International Journal of Computer Vision*, v. 111, n. 1, pp. 98–136, jan 2015. doi: 10.1007/s11263-014-0733-5.

[27] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] KARPATHY, A., FEI-FEI, L. "Deep visual-semantic alignments for generating image descriptions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3128–3137, June 2015. doi: 10.1109/CVPR.2015.7298932.

[29] GRAVES, A., MOHAMED, A., HINTON, G. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649, May 2013. doi: 10.1109/ICASSP.2013.6638947.

[30] CHO, K., VAN MERRIENBOER, B., GULCEHRE, C., et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1179. Disponível em: <http://aclweb.org/anthology/D14-1179>.

[31] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "ImageNet Classification with Deep Convolutional Neural Networks". In: Pereira, F., Burges, C. J. C., Bottou, L., et al. (Eds.), *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., pp. 1097–1105, 2012. Disponível em: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.

[32] NAIR, V., HINTON, G. E. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference*

on International Conference on Machine Learning, ICML'10, pp. 807–814, USA, 2010. Omnipress. ISBN: 978-1-60558-907-7. Disponível em: <http://dl.acm.org/citation.cfm?id=3104322.3104425>.

[33] SUTSKEVER, I., MARTENS, J., DAHL, G., et al. "On the importance of initialization and momentum in deep learning". In: *The 30th International Conference on Machine Learning*, v. 28, Atlanta, Georgia, USA, June 2013.

[34] PASZKE, A., GROSS, S., CHINTALA, S., et al. "Automatic differentiation in PyTorch". 2017.

[35] ABADI, M., AGARWAL, A., BARHAM, P., et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". 2015. Disponível em: <https://www.tensorflow.org/>. Software available from tensorflow.org.

[36] E. HINTON, G., SRIVASTAVA, N., KRIZHEVSKY, A., et al. "Improving neural networks by preventing co-adaptation of feature detectors", *arXiv preprint arXiv:1207.0580*, 2012.

[37] IOFFE, S., SZEGEDY, C. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 448–456. JMLR.org, 2015. Disponível em: <http://dl.acm.org/citation.cfm?id=3045118.3045167>.

[38] SZEGEDY, C., SERMANET, P., REED, S., et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015. doi: 10.1109/CVPR.2015.7298594.

[39] LIN, M., CHEN, Q., YAN, S. "Network in network", *arXiv preprint arXiv:1312.4400*, 2013.

[40] HUANG, G., LIU, Z., V. D. MAATEN, L., et al. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, July 2017. doi: 10.1109/CVPR.2017.243.

[41] XIE, S., GIRSHICK, R., DOLLÁR, P., et al. "Aggregated Residual Transformations for Deep Neural Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5987–5995, July 2017. doi: 10.1109/CVPR.2017.634.

[42] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, June 2016. doi: 10.1109/CVPR.2016.308.

[43] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., et al. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: *ICLR 2016 Workshop*, 2016. Disponível em: <`https://arxiv.org/abs/1602.07261`>.

[44] HOWARD, A. G., ZHU, M., CHEN, B., et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", *arXiv preprint arXiv:1704.04861*, 2017.

[45] SANDLER, M., HOWARD, A. G., ZHU, M., et al. "MobileNetV2: Inverted Residuals and Linear Bottleneck", *arXiv preprint arXiv:1801.04381*, 2018.

[46] DENG, J., DONG, W., SOCHER, R., et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, June 2009. doi: 10.1109/CVPR.2009.5206848.

[47] RUSSAKOVSKY, O., DENG, J., SU, H., et al. "ImageNet Large Scale Visual Recognition Challenge", *International Journal of Computer Vision (IJCV)*, v. 115, n. 3, pp. 211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[48] HUANG, J., RATHOD, V., SUN, C., et al. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3296–3297, July 2017. doi: 10.1109/CVPR.2017.351.

[49] UIJLINGS, J. R. R., VAN DE SANDE, K. E. A., GEVERS, T., et al. "Selective Search for Object Recognition", *International Journal of Computer Vision*, v. 104, n. 2, pp. 154–171, Sep 2013. ISSN: 1573-1405. doi: 10.1007/s11263-013-0620-5. Disponível em: <`https://doi.org/10.1007/s11263-013-0620-5`>.

[50] REN, S., HE, K., GIRSHICK, R., et al. "Object Detection Networks on Convolutional Feature Maps", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 39, n. 7, pp. 1476–1481, July 2017. ISSN: 0162-8828. doi: 10.1109/TPAMI.2016.2601099.

[51] GIRSHICK, R., RADOSAVOVIC, I., GKIOXARI, G., et al. "Detectron". `https://github.com/facebookresearch/detectron`, 2018.

[52] HOCHREITER, S., SCHMIDHUBER, J. "Long Short-term Memory", *Neural computation*, v. 9, pp. 1735–1780, 11 1997. doi: 10.1162/neco.1997.9.8.1735.