OBJECT OF INTEREST DETECTION IN GEOLOGICAL IMAGES WITH DEEP LEARNING

August 2020

ABDOULAYE KOROKO

supervised by

SYLVAIN DESROZIERS

Contents

Chapter 1

INTRODUCTION

1.1 Context

Recent advances in Artificial Intelligence have led to impressive results allowing the development of reliable predictive systems. The use of deep learning and more precisely convolutional neural networks (CNN) [1] has opened up wide prospects and numerous applications have been proposed. Thanks to these techniques, it is now possible for a computer to recognize with certainty the different objects appearing in an image or a film, to identify people precisely or even to analyze documents.

At IFPEN, a very large number of rock samples and other geological data are collected during field missions by geological experts. These collections form an underexploited database. Recent works with CNN led to the implementation of an algorithm allowing the automatic classification of new rock samples from images, intended for geoscience professionals, students or amateur collectors. However to gain a full understanding of images, we should not only focus on the classification of different images, but also try to accurately estimate the concepts and locations of objects contained in each image [2]. This task is known as object detection. Many deep learning based techniques have been proposed for detecting objects from images or videos. In this work, we are particularly interested in those deep learning methods for the detection of objects from digital geological images. These detection algorithms will be applied to two databases for two applications respectively.

1.2 Objectives

1.2.1 Objective 1 : Pre-processing for rock classification

We are seeking to use detection as a pre-processing method before image classification. In fact, work from a recent internship at IFPEN led to the development of a classification algorithm. This algorithm takes as input images of rocks and predicts the class categories of rocks present in the images. This algorithm has two issues which block better accuracy: when a image contains more than one rock or when the rock is badly centered in the image. To resolve this issues, we propose to use a deep learning based detector to automatically crop the images before sending them to the classification algorithm (Figures 1.1 and 1.2).

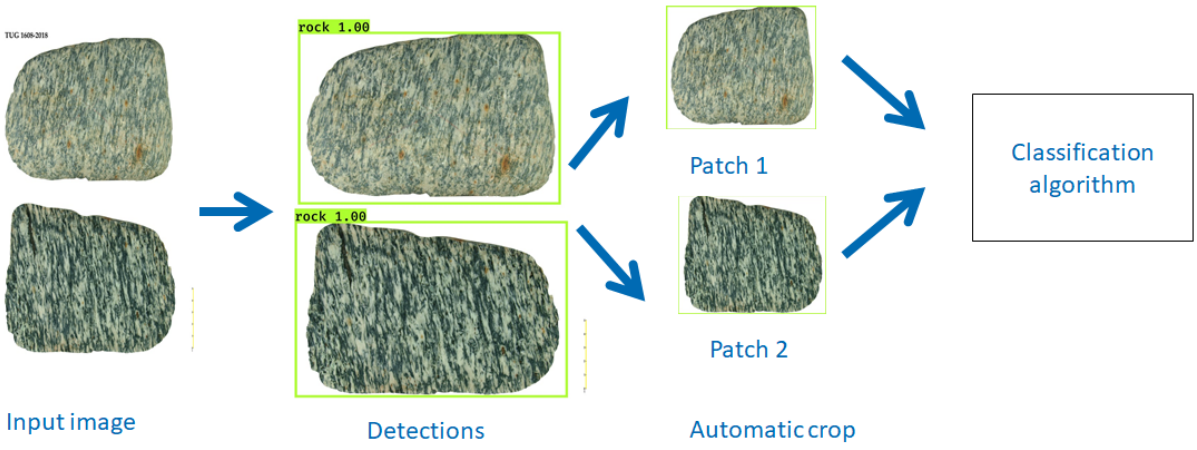


Figure 1.1: Case of more than one rock in an image: the detector detects all the rocks and outputs a patch corresponding to each detected rock. These patches are then successively and independently sent to the classifier for classification.

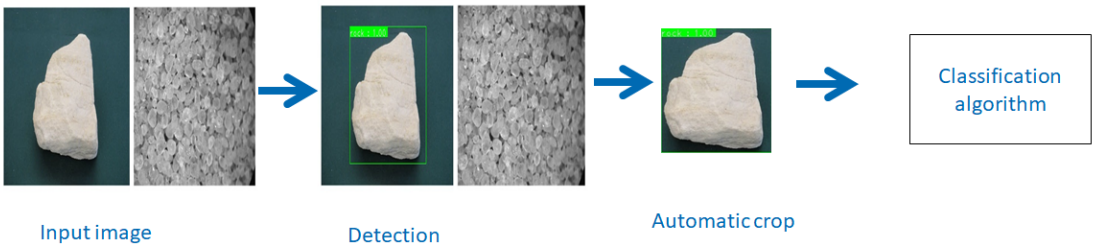


Figure 1.2: Case of a rock badly centered: the detector detects the object of interest (the rock) and outputs a patch containing the rock. The patch is then sent to the classification algorithm.

1.2.2 Objective 2 : Detection of micro-fossils

The second goal of this work is to use the power of CNN for image processing to address the problem of detecting very small objects in images. Indeed, IFPEN has many digital geological images in which micro-fossils, here *foraminifera*, are visible (Figure 1.3). From the rate and type of these micro-fossils, an expert geologist can draw a significant amount of business information. However, the task of manually identifying these micro-fossils in images is tedious work, time-consuming and requires expert geologists. Our goal is to use artificial intelligence methods to automatically identify these foraminifera contained in images (Figure 1.4).

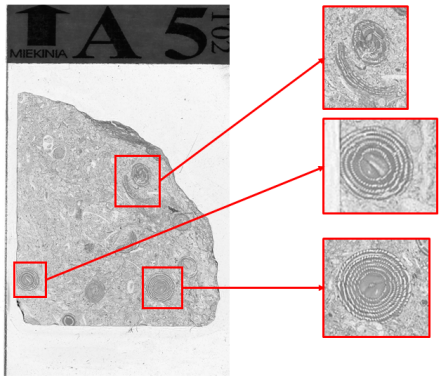


Figure 1.3: Micro-fossils in a geological image

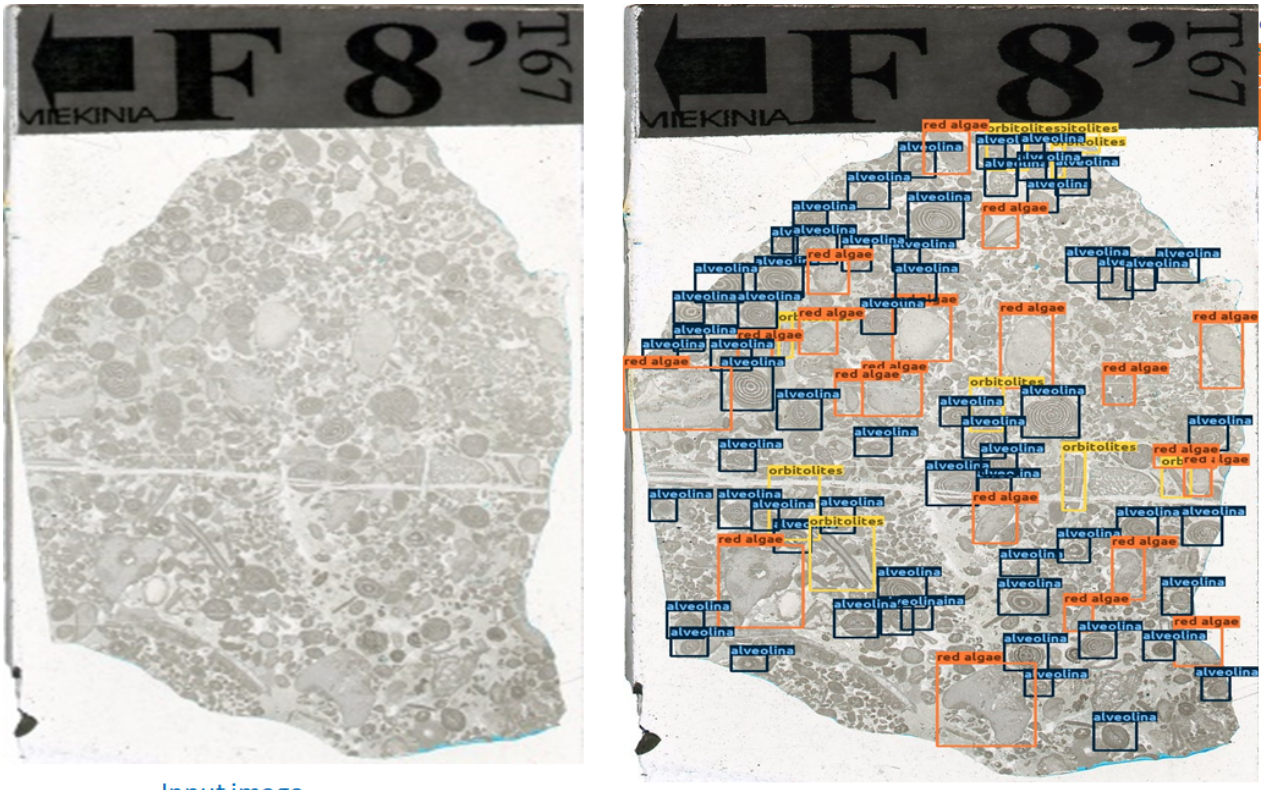


Figure 1.4: Automatic detection of micro-fossils: the IA algorithm takes as input an image and automatically identify micro-fossils contained in the image.

Thus, such an intelligent system could help provide rapid detection for geologists in the field or provide initial image interpretation.

1.3 Strategy

To reach our goals, it will be necessary to explore state of the art deep learning object detection algorithms in order to select the most suitable for our problems. Before exploring the methods, we will firstly introduce the fundamental concepts of deep learning necessary to understand these methods. Secondly data pre-processing will be performed, in particular to ensure that the data labels (fossil positions, etc.) are compatible with different studied algorithms. Finally, we apply the selected methods to our datasets.

Chapter 2

FUNDAMENTAL CONCEPTS

In this chapter, we present fundamental concepts of deep learning useful for understanding object detection algorithms. In the following sections, we will first introduce the basic idea of deep learning, we will then explain multilayer perceptron (MLP) and convolutional neural networks. Finally, we will present the most popular optimization algorithms used in deep learning.

2.1 A brief overview of deep learning and supervised learning

2.1.1 Deep learning

Deep learning (DL) is part of a wider family of machine learning algorithms based on artificial neural networks. It uses multiple neural layers to progressively extract higher level features from the input data. DL has been applied to many fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where it has produced results comparable to and in some cases surpassing human expert performance. DL algorithms can be supervised, semi-supervised or unsupervised. In this report, we will mainly deal with supervised learning.

2.1.2 Supervised learning

Statistical learning denotes the task of learning from data. Supervised learning is a field of statistical learning where data is composed of inputs and outputs (labelled data). The inputs have some influence on outputs and the goal is to learn the correlation between the inputs and outputs in order to be able to predict outputs for future inputs that the model has never seen [3, 4]. In the literature, the inputs are called *predictors*, *independent variables* or *features* while the outputs are named *targets*, *responses* or *dependant variables*. When the outputs are quantitative variables (continue values), supervised learning task is referred as *regression* otherwise (outputs are qualitative values) it is referred as *classification*.

Formally, in supervised learning, we have a training data composed of couples of pair $\{ (\mathrm{x}^{(i)}, \mathrm{y}^{(i)}) \}_{i \in [1,n]}$ where $\forall i, \mathrm{x}^{(i)} \in \mathbb{R}^p,\ p \geq 1,\ \mathrm{y}^{(i)} \in \mathbb{R}$ in case of regression or $\mathrm{y}^{(i)} \in \mathbb{R}^I$ in case of classification (I is the number of class categories). Here n is the number of training examples and p is the dimension of the inputs (very large in deep learning). One assumes that $\forall i,\ \mathrm{y}^{(i)} = f(\mathrm{x}^{(i)}) + \boldsymbol{\varepsilon}^{(i)}$ where $\boldsymbol{\varepsilon}^{(i)}$ denotes a noise and f an unknown function. The principle is to compute an approximation function $g_{\boldsymbol{\theta}}$ of f which minimizes the mean error L :

$$L(\boldsymbol{\theta}, \mathrm{x}, f) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g_{\boldsymbol{\theta}}(\mathrm{x}^{(i)}), f(\mathrm{x}^{(i)}))$$

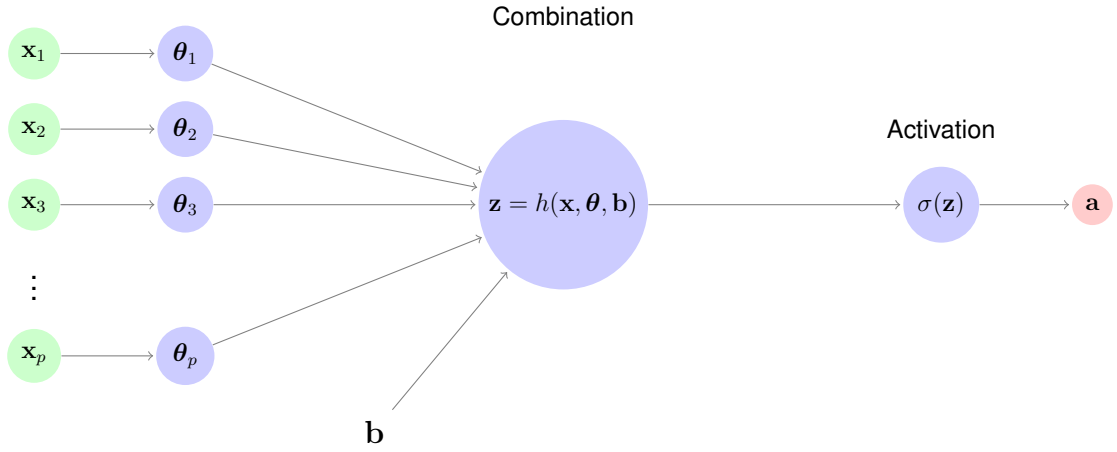
where $\mathcal{L}$ denotes the cost function.

Training process can be seen as an optimization problem. Indeed it consists of finding the best parameters $\boldsymbol{\theta}$ of the estimated function $g_{\boldsymbol{\theta}}$ that minimize the mean error.

2.2 Multilayer perceptron

2.2.1 A perceptron or neuron

A perceptron [6] is a binary classifier: it takes as input a vector $\mathrm{x} \in \mathbb{R}^p$ and computes a binary output a (so called activation). When the activation value is 1, the neuron is said activated otherwise (activation equal to 0) the neuron is said non-activated. The figure below presents the architecture of a neuron.



Here we note:

- $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_p)$ and $\mathbf{b}$ the weights and bias respectively.
- $(\mathrm{x}, \boldsymbol{\theta}, \mathbf{b}) \mapsto h(\mathrm{x}, \boldsymbol{\theta}, \mathbf{b})$ aggregation function :

$$h(\mathbf{x}, \boldsymbol{\theta}, \mathbf{b}) = \sum_{i=1}^{p} \boldsymbol{\theta}_i \, \mathrm{x}_i + \mathbf{b}$$

- $\mathrm{z} \mapsto \sigma(\mathrm{z})$ the activation function.

As shown in the above figure, the general functioning of a neural is as follows : first one applies the aggregation function h to input data x to have z : $\mathrm{z} = h(\mathrm{x}, \boldsymbol{\theta}, \mathbf{b})$ and the output a of the neuron is obtained by applying the activation function to z : $\mathrm{a} = \sigma(\mathrm{z})$.

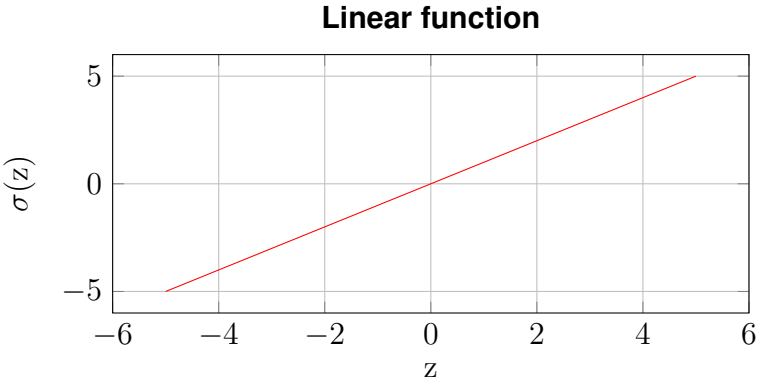
2.2.2 Activation functions

Because they are so crucial for deep learning, before going further, let's take a quick look at some common activation functions. Activation functions are one the most important elements which enable the great successes of deep neural networks [26]. They are differentiable and monotonous functions and have different forms that impact training of the network. In this work, we present the most popular activation functions, their properties and their frames of use. There are 2 types of activation functions: linear functions and non linear functions.

- **Linear Activation function**

It creates an output proportional to the input.

$$\sigma(\mathrm{z}) = \alpha \mathrm{z} \tag{2.1}$$



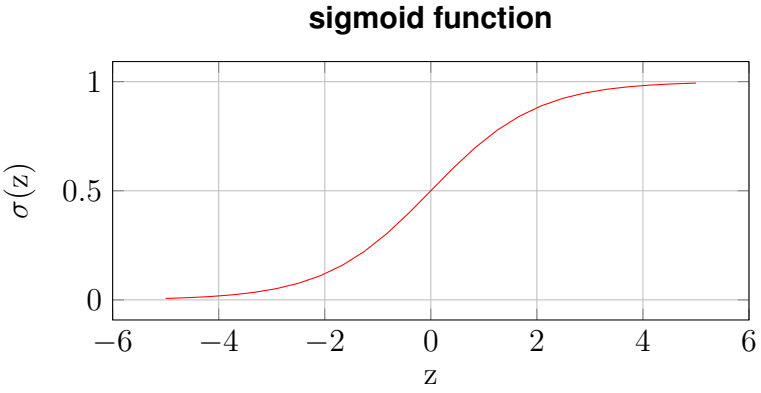
- **Nonlinear Activation Functions**

Non-linear activation functions are the most used activation functions. Such functions allow the model to learn non-linearity, to generalize and differentiate the results. The most popular and used are:

a- Sigmoid or logistic

The main reason why sigmoid function [28] is used is that its output values bound between 0 and 1. As the matter of fact, it is mainly used for models where one has to predict a probability as output (binary classification models). Its expression is:

$$\sigma(\mathrm{z}) = \frac{1}{1 + e^{-\mathrm{z}}} \tag{2.2}$$



The generalization of the logistic function is the softmax function. It is used in the case of a classification with K classes where $K \geq 2$. For a vector $\mathrm{z} = (\mathrm{z}_1, \mathrm{z}_2, ..., \mathrm{z}_K)$ of K real-numbers, softmax is defined by :

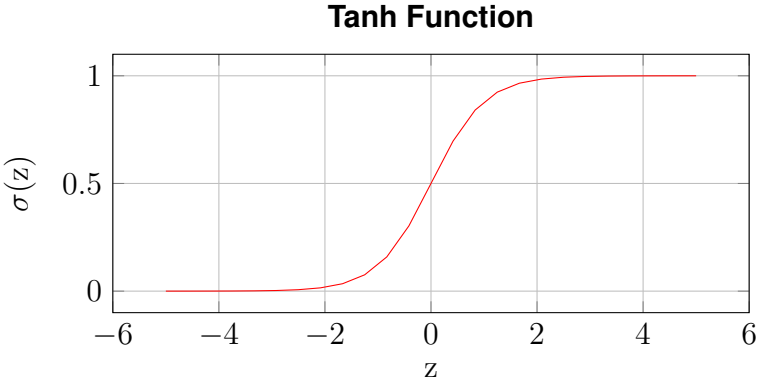
$$\sigma(\mathrm{z})_j = \frac{e^{\mathrm{z}_j}}{\sum_{k=1}^{K} e^{\mathrm{z}_k}} \tag{2.3}$$

softmax function is only used for the output units.

b- Hyperbolic Tangent (Tanh)

Tanh [29] is like sigmoid function but in better version. Its output values fall in the interval [-1,1]. The advantage of such function is that it is zero centered, making it easier to model inputs that have strongly negative, neutral, and strongly positive values. Its mathematical expression is:

$$\sigma(\mathrm{z}) = \frac{1 - e^{-2\mathrm{z}}}{1 + e^{-2\mathrm{z}}} \tag{2.4}$$

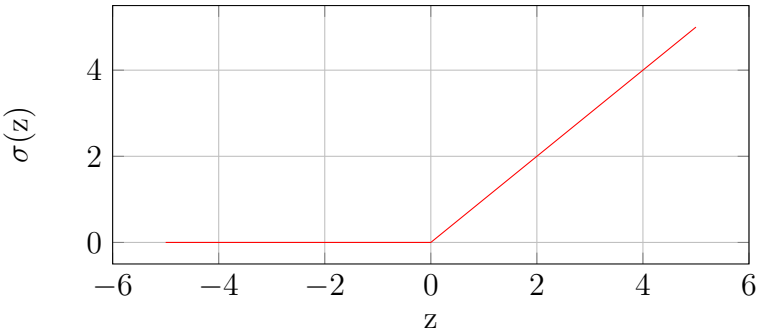


c- ReLU (Rectified Linear Unit)

ReLU [27, 29] is currently the most used activation function in neural networks. It is computationally efficient (allows the model to converge very quickly). It's defined by:

$$\sigma(\mathrm{z}) = max(\mathrm{z}, 0) \tag{2.5}$$

ReLU function

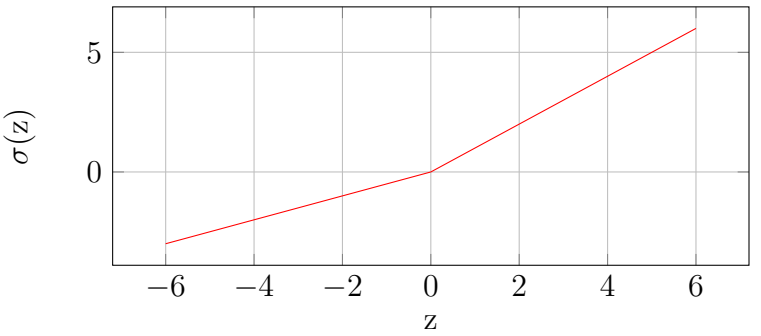


The problem with the ReLU function is that all negative values immediately become zero, which decreases the model's ability to adapt or train correctly from the data. To overcome this problem, one introduced function **Leaky ReLU** [27] whose expression is:

$$\sigma(\mathrm{z}) = \begin{cases} \alpha\mathrm{z} & \text{si} \quad \mathrm{z} < 0 \\ \mathrm{z} & \text{si} \quad otherwise \end{cases} \tag{2.6}$$

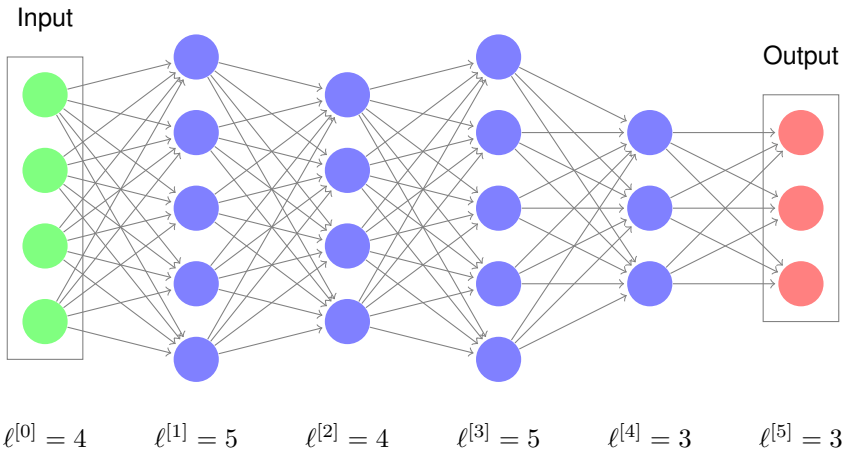
where $\alpha \geq 0$

Leaky ReLU



2.2.3 Definition of Multilayer perceptron

Multi-layer perceptron (MLP) [6] is a type of artificial neural network organized in several layers in which information flows from the input layer to the output layer only. MLP consists of at least three layers: an input layer, a hidden layer and an output layer. Each layer consists of a variable number of perceptrons or neurons. The following figure presents an example of MLP architecture: it contains an input layer, four hidden layers and an output layer.



Except for the input neurons, each neuron uses a nonlinear activation function. Neurons of a hidden layer receive the outputs of neurons of the layer which precedes this layer and send their outputs to the neurons of the next layer through the synaptic weights. Similarly, neurons of the output layer receive the outputs of the last hidden layer as inputs. For a neuron j of a layer ℓ, $\sigma_j^{[\ell]}$ denotes its activation function, $\mathrm{b}_j^{[\ell]}$ its bias and $\boldsymbol{\theta}_{j,k}^{[\ell]}$ connection weights between this neuron j and neurons of previous layer (layer $\ell - 1$). Its output is $\mathrm{a}_j^{[\ell]} = \sigma_j^{[\ell]}(\mathrm{z}_j^{[\ell]})$ with $\mathrm{z}_j^{[\ell]} = \sum_k \boldsymbol{\theta}_{j,k}^{[\ell]}\, \mathrm{a}_k^{[\ell-1]} + \mathrm{b}_j^{[\ell]}$.
In the following, we consider the notations:

- x input data
- y target
- $\mathcal{L}$ loss function
- ℓ_o total number of layers
- $\boldsymbol{\theta}^{[\ell]} = (\boldsymbol{\theta}_{j,k}^{[\ell]})$ the weights matrix between layer between layers $\ell - 1$ and ℓ.

- $\mathbf{b}^{[\ell]}$ bias of layer ℓ
- $\mathrm{z}^{[\ell]}$ weighted input of layer ℓ
- $\sigma^{[\ell]}$ activation functions of layer ℓ
- $\mathrm{a}^{[\ell]} = \sigma^{[\ell]}(\mathrm{z}^{[\ell]})$ activations of layer ℓ

2.2.4 Forward propagation and backward propagation

2.2.4.1 Forward propagation

MLP belongs to a specific family of neural networks called feed-forward neural networks. In such neural networks, to compute the output, the input data should be fed in the forward direction only. The data should not flow in reverse direction during output computation otherwise it would form a cycle and the output could never be computed.

Forward propagation (FP) consists of feeding the input data in the forward direction through the network. FP can be seen as matrix multiplication. Let's consider an input data $\mathrm{x} \in \mathbb{R}^{p \times n}$ where p and n denote dimension of inputs and number of training examples respectively. FP refers to sequential multiplication of x by weights matrix and then application of activation functions from the first hidden layer to output layer. In the case of the MLP above, FP is the following operations:

1. $\mathrm{z}^{[1]} = \boldsymbol{\theta}^{[1]}\mathrm{x} + \mathbf{b}^{[1]}$

$\mathrm{a}^{[1]} = \sigma^{[1]}(\mathrm{z}^{[1]})$

2. $\mathrm{z}^{[2]} = \boldsymbol{\theta}^{[2]}\mathrm{a}^{[1]} + \mathbf{b}^{[2]}$

$\mathrm{a}^{[2]} = \sigma^{[2]}(\mathrm{z}^{[2]})$

3. $\mathrm{z}^{[3]} = \boldsymbol{\theta}^{[3]}\mathrm{a}^{[2]} + \mathbf{b}^{[3]}$

$\mathrm{a}^{[3]} = \sigma^{[3]}(\mathrm{z}^{[3]})$

4. $\mathrm{z}^{[4]} = \boldsymbol{\theta}^{[4]}\mathrm{a}^{[3]} + \mathbf{b}^{[4]}$

$\mathrm{a}^{[4]} = \sigma^{[4]}(\mathrm{z}^{[4]})$

5. $\mathrm{z}^{[5]} = \boldsymbol{\theta}^{[5]}\mathrm{a}^{[4]} + \mathbf{b}^{[5]}$

$$\mathrm{a}^{[5]} = \hat{\mathrm{y}} = \sigma^{[5]}(\mathrm{z}^{[5]})$$

Generally, in a feed-forward neural network with ℓ_o layers, FP is denoted by the following operations:

$$\hat{\mathrm{y}} = \mathrm{a}^{[\ell_o]} = \sigma^{[\ell_o]}\left(\boldsymbol{\theta}^{[\ell_o]}\sigma^{[\ell_o-1]}\left(\boldsymbol{\theta}^{[\ell_o-1]}\cdots\sigma^{[1]}\left(\boldsymbol{\theta}^{[1]}\mathrm{x}+\mathbf{b}^{[1]}\right)\cdots+\mathbf{b}^{[\ell_o-1]}\right)+\mathbf{b}^{[\ell_o]}\right)$$

2.2.4.2 Backward propagation

Back propagation (BP) refers to the method of computing the gradient of the loss function with respect to the parameters (weights and bias) of the network. BP is the generalization of the *delta rule* [6]. In this paper, we do not treat bias since they correspond to a weight with a fixed input of 1. In BP, the forms of loss function and the activation functions do not matter since they and their derivatives can be evaluated effectively. With the assumption of not taking into account bias, the output $\hat{\mathrm{y}}$ of the network given an input x is :

$$\hat{\mathrm{y}} = \mathrm{a}^{[\ell_o]} = \sigma^{[\ell_o]}\left(\boldsymbol{\theta}^{[\ell_o]}\sigma^{[\ell_o-1]}\left(\boldsymbol{\theta}^{[\ell_o-1]}\cdots\sigma^{[1]}\left(\boldsymbol{\theta}^{[1]}\mathrm{x}\right)\cdots\right)\right)$$

Let's consider a training set of input-output pairs

$$\mathcal{T} = \left\{\left(\mathrm{x}^{(1)},\mathrm{y}^{(1)}\right),\left(\mathrm{x}^{(2)},\mathrm{y}^{(2)}\right),\cdots,\left(\mathrm{x}^{(n)},\mathrm{y}^{(n)}\right)\right\}.$$

For each pair $\left(\mathrm{x}^{(i)},\mathrm{y}^{(i)}\right)$, $\mathcal{L}\left(\mathrm{x}^{(i)},\mathrm{y}^{(i)}\right)$ denotes the loss between network's prediction and the actual value. BP computes the gradient of the loss for a fixed pair $\left(\mathrm{x}^{(i)},\mathrm{y}^{(i)}\right)$ where weights $\boldsymbol{\theta}^{[\ell]}_{j,k}$ can vary. Each component $\partial\mathcal{L}/\partial\boldsymbol{\theta}^{[\ell]}_{j,k}$ can be individually computed by chain rule as treated in [6]. However this computation procedure is not efficient. BP efficiently computes the gradient of the loss with respect to the weighted input of each layer ($\delta^{[\ell]} = \partial\mathcal{L}/\partial\mathrm{z}^{[\ell]}$) from back to front. The gradient of the weights in layer ℓ is then:

$$\partial\mathcal{L}/\partial\boldsymbol{\theta}^{[\ell]} = \nabla_{\boldsymbol{\theta}^{[\ell]}}\mathcal{L} = \delta^{[\ell]}\left(\mathrm{a}^{[\ell-1]}\right)^T$$

Indeed, by chain rule :

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}^{[l]}} = \frac{\partial\mathcal{L}}{\partial\mathrm{z}^{[\ell]}}\cdot\frac{\partial\mathrm{z}^{[\ell]}}{\partial\boldsymbol{\theta}^{[l]}} = \delta^{[\ell]}\left(\mathrm{a}^{[\ell-1]}\right)^T$$

Let's denote $\left(\sigma^{[\ell]}\right)'\left(\mathrm{z}^{[\ell]}\right)$ by $\left(\sigma^{[\ell]}\right)'$ for all ℓ. Now by chain rule :

$$\begin{aligned}\delta^{[\ell_o]} &= \frac{\partial \mathcal{L}}{\partial \mathrm{z}^{[\ell_o]}} \\ &= \frac{\partial \mathcal{L}}{\partial \mathrm{a}^{[\ell_o]}} \cdot \frac{\partial \mathrm{a}^{[\ell_o]}}{\partial \mathrm{z}^{[\ell_o]}} \\ &= \left(\sigma^{[\ell_o]}\right)' \cdot \nabla_{a^{[\ell_o]}} \mathcal{L}\end{aligned}$$

$\nabla_{\mathrm{a}^{[\ell_o]}} \mathcal{L}$ can be easily computed. Thus, $\forall \ell \in \{1, \ldots, \ell_o - 1\}$:

$$\begin{aligned}\delta^{[\ell]} &= \frac{\partial \mathcal{L}}{\partial \mathrm{z}^{[\ell]}} \\ &= \frac{\partial \mathcal{L}}{\partial \mathrm{z}^{[\ell+1]}} \cdot \frac{\partial \mathrm{z}^{[\ell+1]}}{\partial \mathrm{z}^{[\ell]}} \\ &= \left(\sigma^{[\ell]}\right)' \cdot \left(\boldsymbol{\theta}^{[\ell+1]}\right)^T \cdot \delta^{\ell+1}\end{aligned}$$

This recursive relationship allows to compute the gradient of the weights of each layer from back to front :

$$\begin{aligned}\delta^{[1]} &= \left(\sigma^{[1]}\right)' \cdot \left(\boldsymbol{\theta}^{[2]}\right)^T \cdot \left(\sigma^{[2]}\right)' \cdots \left(\boldsymbol{\theta}^{[\ell_o-1]}\right)^T \cdot \left(\sigma^{\ell_o-1}\right)' \cdot \left(\boldsymbol{\theta}^{[\ell_o]}\right)^T \cdot \left(\sigma^{[\ell_o]}\right)' \cdot \nabla_{\mathrm{a}^{[\ell_o]}} \mathcal{L} \\ \delta^{[2]} &= \left(\sigma^{[2]}\right)' \cdots \left(\boldsymbol{\theta}^{[\ell_o-1]}\right)^T \cdot \left(\sigma^{[\ell_o-1]}\right)' \cdot \left(\boldsymbol{\theta}^{[\ell_o]}\right)^T \cdot \left(\sigma^{[\ell_o]}\right)' \cdot \nabla_{\mathrm{a}^{[\ell_o]}} \mathcal{L} \\ &\vdots \\ \delta^{[\ell_o-1]} &= \left(\sigma^{[\ell_o-1]}\right)' \cdot \left(\boldsymbol{\theta}^{[\ell_o]}\right)^T \cdot \left(\sigma^{[\ell_o]}\right)' \cdot \nabla_{\mathrm{a}^{[\ell_o]}} \mathcal{L} \\ \delta^{[\ell_o]} &= \left(\sigma^{[\ell_o]}\right)' \cdot \nabla_{a^{[\ell_o]}} \mathcal{L}\end{aligned}$$

2.2.5 Training

Feed-forward neural networks are trained with *gradient descent based algorithms*. During training, forward and backward propagation depend on each other. In particular, forward propagation computes the loss between the predictions and actual values. Then back propagation computes the gradient of the loss with respect to the weights. Finally, the weights are updated with a gradient descent based optimization algorithm. This procedure is repeated a number of times or iterations. In the section 2.5, we will present the most used algorithms to update the model weights.

2.3 Convolutional neural networks

2.3.1 Motivation

Multilayer perceptrons are less accurate and time consuming when it comes to deal with unstructured data (images, videos). Their number of parameters depend on the dimension of input data. For instance, for a simple image binary classification task (dog or cat) with MLP, one can have a model with 10^9 parameters. Unless we have an extremely large dataset, lots of GPUs, and an extraordinary amount of patience, learning the parameters of this network may turn out to be impossible [30, 5]. CNN based on mathematical convolution are convenient for such data since the size of their parameters does not depend on input data dimension. CNN are the core element that has revolutionized deep learning and in particular computer vision. They are multi-layered artificial neural networks specialized on recognizing visual patterns directly from image pixels. (LeCun et al. 1990) proved the effectiveness of CNN in entity recognition in images.

2.3.2 Mathematical expressions of convolutional operations

Here we recall mathematical convolution operations.
The convolution between two functions $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined by :

$$[\, f \circledast g \,](\mathrm{x}) = \int_{\mathbb{R}^d} f(\mathrm{z})\, g(\mathrm{x}-\mathrm{z})\, d\,\mathrm{z}$$

In case of discrete measure, for example measure that has $\mathbb{Z}$ as support, convolution is defined as :

$$[\, f \circledast g \,](i) = \sum_{a \in \mathbb{Z}} f(a)\, g(i-a)$$

Convolution between two matrix $\mathcal{X} \in \mathbb{R}^{n_x \times p_x}$, $\mathcal{F} \in \mathbb{R}^{n_f \times p_f}$ is a matrix $\mathcal{R} \in \mathbb{R}^{n_r \times p_r}$ with $n_r = n_x - n_f + 1$, $p_r = p_x - p_f + 1$ and

$$\forall (i,j),\ \mathcal{R}(i,j) = \sum_{n=0}^{n_x-1} \sum_{p=0}^{p_x-1} \mathcal{X}(n,p)\, \mathcal{F}(i-n, j-p)$$

2.3.3 Convolution between images and filters

A numerical image is represented by a matrix (2D array for gray scale and 3D for RGB). The mathematical convolution operation between two matrix is used to process

images (edge and contour detection, smoothing, thresholding, etc). The technique consists in convolving images with filters. In image processing, the third dimension of the image (1 for gray scale and 3 for RGB) is called *channel*. To perform convolution between an image and a filter, it is necessary that the image and filter have the same channel. Several filters can be applied to the same image to detect more features. In that case, the output is the concatenation of the different results of the convolution of the input image with each filter (Figures 2.1, 2.2, 2.3). When we make convolution between an image of shape $\mathrm{n} \times \mathrm{n} \times \mathrm{c}$ with a filter of shape $\mathrm{f} \times \mathrm{f} \times \mathrm{c}$, we obtain a feature matrix of shape $\lfloor \mathrm{n} - \mathrm{f} + 1 \rfloor \times \lfloor \mathrm{n} - \mathrm{f} + 1 \rfloor$.

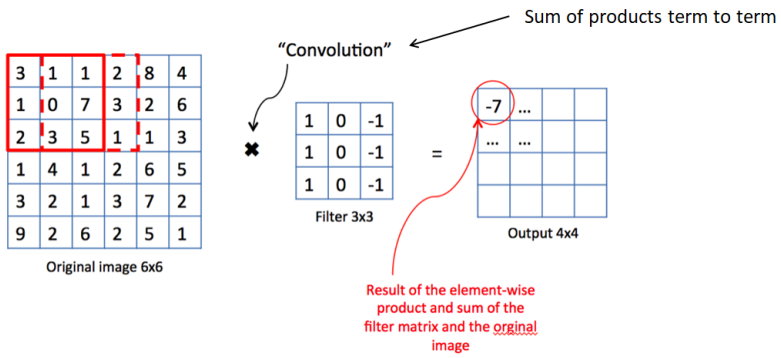


Figure 2.1: Convolution between a 2D image and a 2D filter

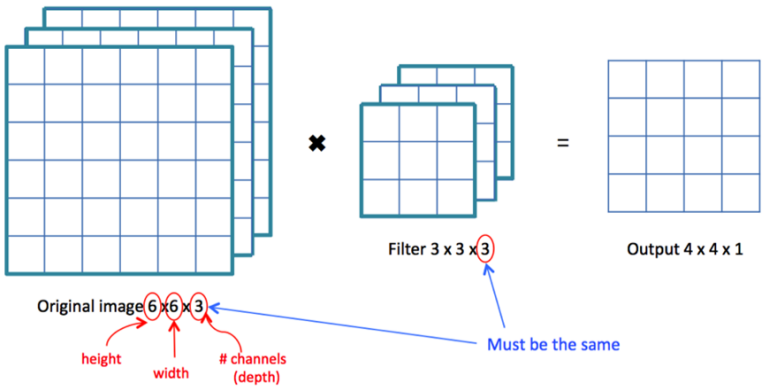


Figure 2.2: convolution between a 3D image and a 3D filter

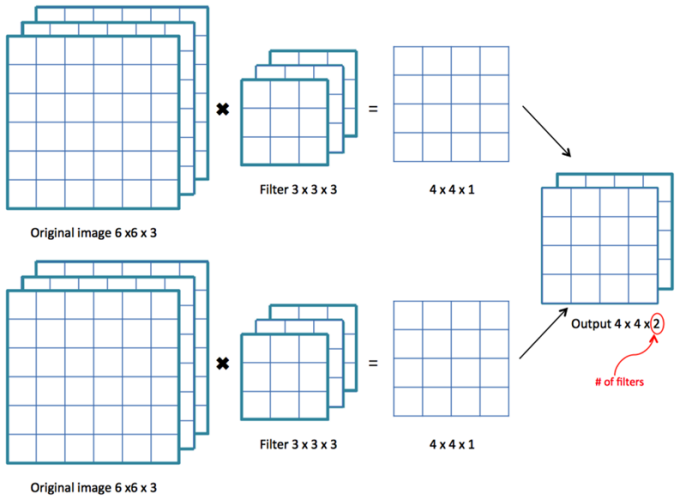


Figure 2.3: convolution between a 3D image and two 3D filters

2.3.4 Padding and strided convolution

- **padding** convolution shrinks output and therefore throws away a lot of information that are in the edges. To solve these problems we can use padding before convolution that refers to adding some rows and columns of 0 around the boundary of the input image. The convolution between an image of shape $\mathrm{n} \times \mathrm{n} \times \mathrm{c}$ and a filter of shape $\mathrm{f} \times \mathrm{f} \times \mathrm{c}$ with a padding p outputs a matrix of shape $\lfloor \mathrm{n} + 2\mathrm{p} - \mathrm{f} + 1 \rfloor \times \lfloor \mathrm{n} + 2\mathrm{p} - \mathrm{f} + 1 \rfloor$.

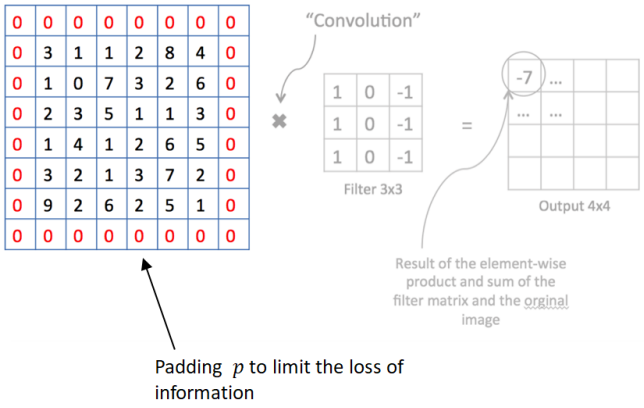


Figure 2.4: Padding $\mathrm{p} = 1$ before convolution

- **strided convolution** sometimes, because of computational efficiency or down-sampling, a stride s is used as the number of pixels we will jump when we are convolving filter. The convolution between an image of shape $\mathrm{n} \times \mathrm{n} \times \mathrm{c}$ and a filter of shape $\mathrm{f} \times \mathrm{f} \times \mathrm{c}$ with strides s and s' for row and column respectively outputs a feature map of shape $\left\lfloor \frac{\mathrm{n}-\mathrm{f}}{\mathrm{s}} + 1 \right\rfloor \times \left\lfloor \frac{\mathrm{n}-\mathrm{f}}{\mathrm{s}'} + 1 \right\rfloor$.

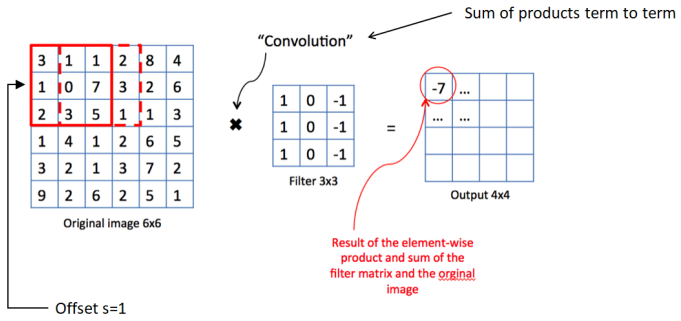


Figure 2.5: Convolution with strides $\mathrm{s} = 1$ and $\mathrm{s}' = 1$

Finally, if a matrix image $\mathrm{n} \times \mathrm{n} \times \mathrm{c}$ is convolved with $\mathrm{f} \times \mathrm{f} \times \mathrm{c}$ filter, padding p strides s and s', we obtain a matrix using the following general shape formula

$$\left\lfloor \frac{\mathrm{n} + 2\mathrm{p} - \mathrm{f}}{\mathrm{s}} + 1 \right\rfloor \times \left\lfloor \frac{\mathrm{n} + 2\mathrm{p} - \mathrm{f}}{\mathrm{s}'} + 1 \right\rfloor$$

.

2.3.5 Convolution layer

Given an input x, a convolutional layer refers to the following operation steps :

- Apply a padding p (optional)
- Choose a stride s
- Choose dimension and number of filters
- Convolve x with each of the filters and concatenate the results to have the output.
- Add a bias (optional)
- Apply an activation function (ReLU)

A convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects. An important note is that the parameters of a convolution layer do not depend on the dimension of the input but only on the sizes and numbers of filters used and possibly the biases. A convolution layer which contains n_f filters of shape $\mathrm{f} \times \mathrm{f} \times \mathrm{c}$ has $\mathrm{n}_f \times \{\mathrm{f} \times \mathrm{f} \times \mathrm{c}\} + \mathrm{n}_f$ parameters.

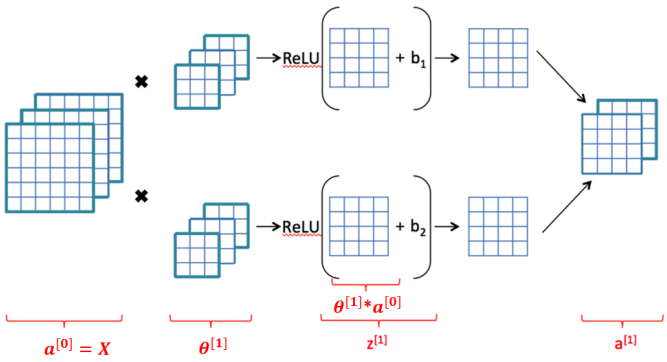


Figure 2.6: A convolution layer (convolution + ReLU)

2.3.6 Pooling layer

Pooling layers [31] are often used after convolution layers to reduce the size of the inputs, speed up computation, and to make features detection more robust. Pooling operators consist of a fixed-shape window that is slided over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window. There exits two kind of pooling layer:

- **Max pooling** is saying, if the feature is detected anywhere in this filter then keep a high number.
- **Average pooling** is taking the averages of the values instead of taking the max values.

Pooling layers have no parameters to learn. They only have hyper-parameters: window size f, stride s, max or average pooling.

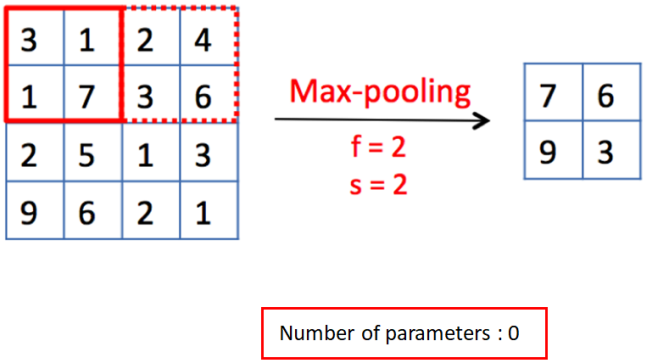


Figure 2.7: Max pooling

2.3.7 Popular Convolutional Neural Networks

CNN consist to putting all of the previous tools together. Precisely, CNN consist of two parts : (i) a block of convolutional layers followed by pooling layers; and (ii) a block of fully-connected layers.

CNN are trained like MLP with back-propagation and gradient descent.

In the following, we present some examples of popular CNN.

2.3.7.1 LeNet5

LeNet5 [9] is the first published convolutional neural network. It was introduced by Yann LeCun on the purpose of handwritten digits recognition. It has 60k parameters.

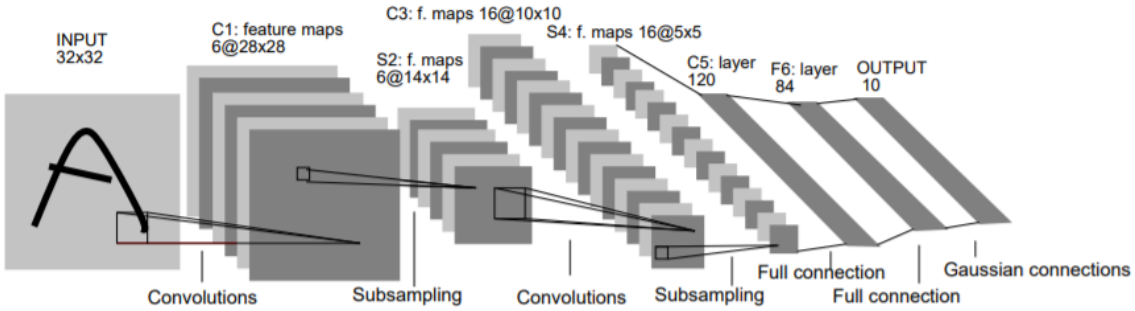


Figure 2.8: Architecture of LeNet5. The input is a handwritten digit image, the output a probabilitiy over 10 possible classes.

2.3.7.2 AlexNet

AlexNet [10] was introduced in 2012 and won the ImageNet Large Scale Visual Recognition Challenge 2012. The top 5 score was 15.3% which was an impressive improvement far from others networks. It has 60 Million parameters and was a first implementation of a network on GPU. AlexNet had a large impact on the field of machine learning for computer vision.

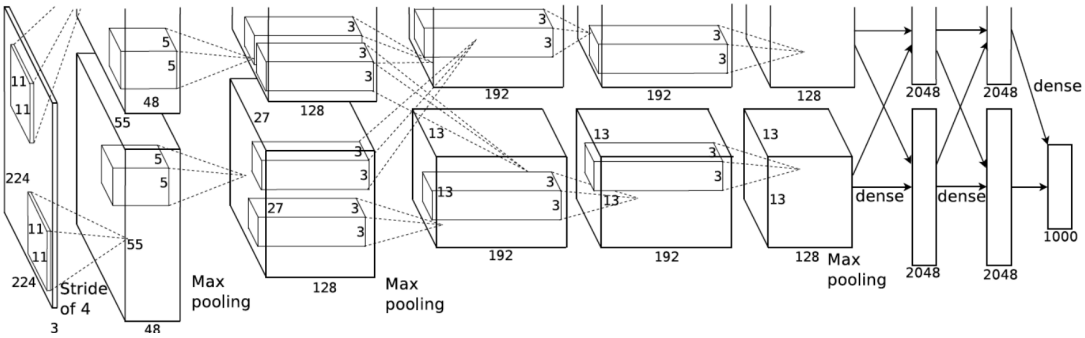


Figure 2.9: Architecture of AlexNet. The input is a RGB image, the output a probabilitiy over 1000 possible classes.

2.3.7.3 VGG-16

VGG-16 is a deep CNN model introduced by K. Simonyan and A. Zisserman in [11]. Considered to be one of the excellent CNN architecture, VGG-16 has 16 weight layers (13 convolutional layers, 5 Max Pooling layers and 3 dense layers) and was used to win ImageNet competition in 2014. VGG-16 has around 138 million parameters. There are another version named VGG-19 which is a bigger version (19 weight layers). Nevertheless most people uses VGG-16 instead of VGG-19 because both do the same work [8].

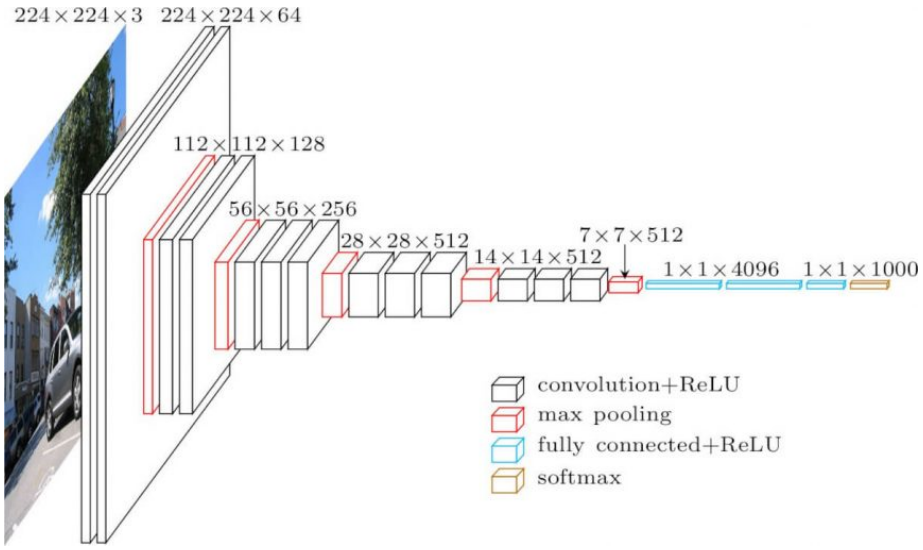


Figure 2.10: Architecture of VGG-16

2.3.7.4 Residual Networks (ResNet)

Very deep neural networks are hard to train because of vanishing and exploding gradient problems [12]. To solve this problem, residual blocks [12] was introduced by Microsoft researchers. They refer to skip connections or shortcuts (Figure 2.12) which allow you to take the activation from one layer and suddenly feed it to another layer even much deeper in neural network which allows you to train large neural networks even with layers greater than 100 [8].

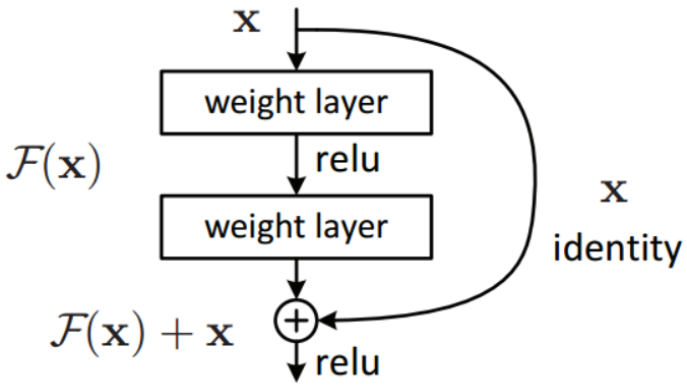


Figure 2.11: Residual learning: a building block

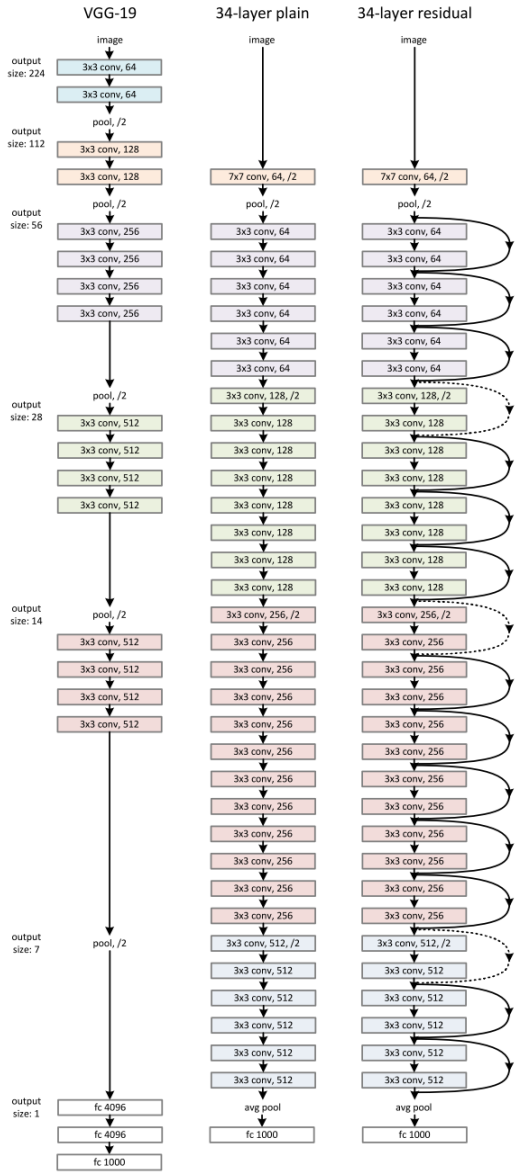


Figure 2.12: **Left**: the VGG-19 model (19.6 billion FLOPs(multiply-adds)) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs)[12]

2.4 Improving Deep Neural Networks : Regularization and Batch Normalization

When we are dealing with machine learning algorithms (here neural networks), we can encounter to main issues: *underfitting* and *overfitting*.

We encounter *underfitting* when the model is not capable of properly capturing the information in the data (the model does poorly fit the data). It usually occurs when the data used to train the model is not enough or when we have a lot of training data but the model is not trained enough or when the model is not adapted to the data (for instance a linear model used to fit a non-linear data). In such cases, predictions on training data are wrong and it's said that the model has a high bias. To avoid underfitting, it's necessary to have a lot of training data, train enough the model and also select a model adapted to the data (models selection).

An *overfitted* model is a model that is too specialized on the training data and that will not be generalized well. Instead of learning the correlation between features, the model will memorize the training data. Therefore, it makes good predictions on training data (high accuracy) and has wrong predictions on test data. In that case the model has high variance. Overfitting usually occurs with complex models (deep neural networks, non parametric and non-linear algorithms).

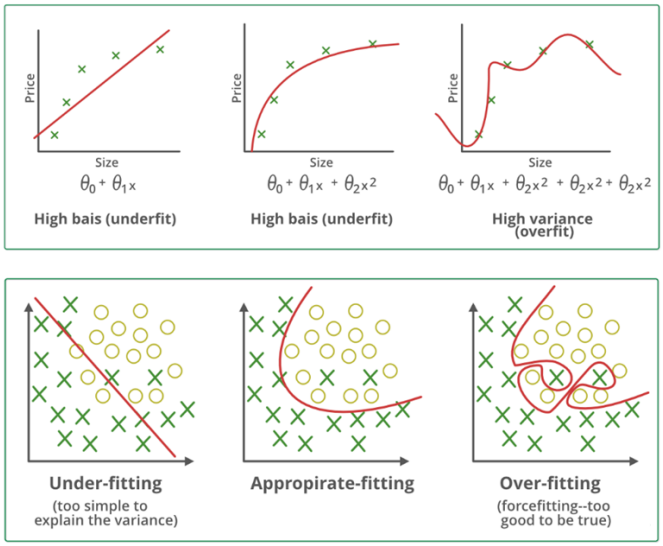


Figure 2.13: Example of underfitting-overfitting in regression and classification

In this section, we present the most popular techniques used to avoid overfitting in

deep learning.

2.4.1 Regularization

Regularization refers to the technique of adding a penalty to the cost function with the aim of avoiding overfitting (reduce variance). It consists of replacing the classical cost

$$L = \frac{1}{n} \sum_{j=1}^{n} \mathcal{L}\left(\hat{\mathrm{y}}_j, \mathrm{y}_j\right)$$

by

$$L = \frac{1}{n} \sum_{j=1}^{n} \mathcal{L}\left(\hat{\mathrm{y}}_j, \mathrm{y}_j\right) + \mathcal{P}\left(\boldsymbol{\theta}\right)$$

where $\mathcal{P}\left(\boldsymbol{\theta}\right)$ denotes a penalization function. The L2 regularization technique which uses Frobenius norm as penalization function is the most used :

$$\mathcal{P}(\boldsymbol{\theta}) = \frac{\lambda}{2n} \sum_{\ell=1}^{\ell_o} \|\boldsymbol{\theta}^{[\ell]}\|_F^2$$

where

$$\forall \boldsymbol{\theta}^{[\ell]} \in \mathbb{R}^{q^{[\ell]} \times p^{[\ell]}},\ \|\boldsymbol{\theta}^{[\ell]}\|_F = \sqrt{\sum_{i=1}^{q^{[\ell]}} \sum_{j=1}^{p^{[\ell]}} \boldsymbol{\theta}_{ij}^{[\ell]2}}$$

is Frobenius or euclidean norm. λ is the regularization parameter (hyperparameter for the network). If λ is well chosen, it will just reduce some weights that make the neural network overfit. However, if λ is too large, a lot of weights will be closed to zeros which will make the network simpler like logistic regression.

2.4.2 Batch Normalization

One of the most important ideas in improving deep neural networks has been an algorithm known as *batch normalization* (BN). BN is a technique used to normalize the input layer by re-centering and re-scaling [13]. More precisely BN refers to the technique of normalizing for any hidden layer ℓ, $\mathrm{a}^{[\ell]}$ or $\mathrm{z}^{[\ell]}$ to train $\boldsymbol{\theta}^{[\ell]}$ and $\mathrm{b}^{[\ell]}$ faster. BN improves the speed, performance, and stability of deep neural networks. The reason why BN works well for deep neural networks is that it reduces internal covariate shift. Internal covariate shift occurs when parameter initialization and changes

in the distribution of the inputs of each layer affect the learning rate of the network [13]. BN also smooths the objective function. BN is usually applied with mini-batches.

Algorithm 1: Batch normalization

Input: Layer ℓ with d-dimensional input, batch $\mathcal{B} = (\mathrm{z}^{(1)}, \cdots, \mathrm{z}^{(m)})$ with $\mathrm{z}^{(i)} \in \mathbb{R}^d$, parameters $\alpha \in \mathbb{R}^d$, $\beta \in \mathbb{R}^d$ and $\varepsilon \in \mathbb{R}$.

Output: Normalized batch $\tilde{\mathcal{B}} = (\tilde{\mathrm{z}}^{(1)}, \cdots, \tilde{\mathrm{z}}^{(1)})$

Compute $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \mathrm{z}^{(i)}$;

Compute $\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(z^{(i)} - \mu_{\mathcal{B}})^2$;

for $i = 1$ *to* m **do**

  for $j = 1$ *to* d **do**

    $z^{(i)}_{\text{norm},j} = \frac{z^{(i)}_j - \mu_{\mathcal{B},j}}{\sqrt{\sigma^2_{\mathcal{B},j} + \varepsilon}}$;

    $\tilde{z_j}^{(i)} = \alpha_j z^{(i)}_{\text{norm},j} + \beta_j$;

return $\tilde{\mathcal{B}} = (\tilde{\mathrm{z}}^{(1)}, \cdots, \tilde{\mathrm{z}}^{(1)})$

α and β are learnable parameters of the model and ε a small number whose role is to avoid dividing by zero.

2.5 Optimization Algorithms

2.5.1 Relationship between optimization and deep learning

For a deep learning problem, a loss function will usually be defined. After the definition of the loss function, an optimization algorithm is used in attempt to minimize the loss. Formally, given a training set

$$\mathcal{T} = \left\{ \left(\mathrm{x}^{(1)}, \mathrm{y}^{(1)}\right), \left(\mathrm{x}^{(2)}, \mathrm{y}^{(2)}\right), \cdots, \left(\mathrm{x}^{(n)}, \mathrm{y}^{(n)}\right) \right\}$$

of n pairs of feature-label vectors, where $\mathrm{x}^{(i)} \in \mathbb{R}^{d_x}$ and $\mathrm{y}^{(i)} \in \mathbb{R}^{d_y}$, a deep learning model can bee seen as a mapping function $F(\cdot; |\boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are the feature and label space respectively and $\boldsymbol{\theta}$ denotes the parameter vector of the mapping function. Given a pair $(\mathrm{x}^{(i)}, \mathrm{y}^{(i)})$, the error between the model prediction $\hat{\mathrm{y}}^{(i)} = F(\mathrm{x}^{(i)}; \boldsymbol{\theta})$ and the actual value $\mathrm{y}^{(i)}$ is denoted by the loss term $\mathcal{L}(\mathrm{y}^{(i)}, \hat{\mathrm{y}}^{(i)})$.

There exists several types of loss functions. The most popular are :

- *Mean Square Error (MSE):*

$$\mathcal{L}_{MSE}\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) = \frac{1}{d_y}\sum_{j=1}^{d_y}\left(\mathbf{y}_i(j) - \hat{\mathbf{y}}_i(j)\right)^2$$

- *Mean Absolute Error (MAE):*

$$\mathcal{L}_{MAE}\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) = \frac{1}{d_y}\sum_{j=1}^{d_y}\left|\mathbf{y}_i(j) - \hat{\mathbf{y}}_i(j)\right|$$

- *Hinge Loss:*

$$\mathcal{L}_{\text{Hinge}}\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) = \sum_{j \neq l_{y_i}} \max\left(0, \hat{\mathbf{y}}_i(j) - \hat{\mathbf{y}}_i\left(l_{\mathbf{y}_i}\right) + 1\right)$$

  where l_{y_i} denotes the true class label index for the instance according to vector y_i.

- *Cross Entropy Loss:*

$$\mathcal{L}_{CE}\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) = -\sum_{j=1}^{d_y}\mathbf{y}_i(j)\log\hat{\mathbf{y}}_i(j)$$

The total loss is then represented by :

$$L(\boldsymbol{\theta}; \mathcal{T}) = \sum_{(\mathbf{x}_i, \mathbf{y}_i)\in\mathcal{T}}\mathcal{L}\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) = \sum_{(\mathbf{x}_i, \mathbf{y}_i)\in\mathcal{T}}\mathcal{L}\left(F\left(\mathbf{x}_i; \boldsymbol{\theta}\right), \mathbf{y}_i\right)$$

Training the deep learning model consists of minimizing the total loss. Formally, it consists of solving the following optimization problem:

$$\underset{\boldsymbol{\theta}\in\Theta}{\operatorname{argmin}}\, L(\boldsymbol{\theta}; \mathcal{T}),$$

where $\boldsymbol{\theta}$ denotes the parameter domain and can be equal to $\mathbb{R}^{d_{\boldsymbol{\theta}}}$ where $d_{\boldsymbol{\theta}}$ is the dimension of vector $\boldsymbol{\theta}$.

To solve this problem, many optimization algorithms based on gradient descent have been introduced. These algorithms can compute the globally optimal or locally optimal parameters for a deep learning model in the case the loss function is convex or non-convex respectively.

2.5.2 Gradient Descent

Gradient descent is an iterative method for minimizing an objective function f defined in a space $\mathcal{E}$ with a norm $\|.\|$. Compared with the high-order derivative based algorithms, e.g. Newton's method, etc., gradient descent is based on first order derivative and is much more efficient and practical for deep learning models.

Algorithm 2: Gradient descent

Input: initial point $\boldsymbol{\theta}^{(0)} \in \mathcal{E}$, maximum number of iterations K, learning rate η
Output: global or local minimum of f

for $k = 1$ *to* K **do**
  Compute $\nabla_{\boldsymbol{\theta}} f\left(\boldsymbol{\theta}^{(k-1)}\right)$ the gradient of f at $\boldsymbol{\theta}^{(k-1)}$;
  Compute next iterate $\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \nabla_{\boldsymbol{\theta}} f\left(\boldsymbol{\theta}^{(k-1)}\right)$
return $\boldsymbol{\theta}^{(K)}$

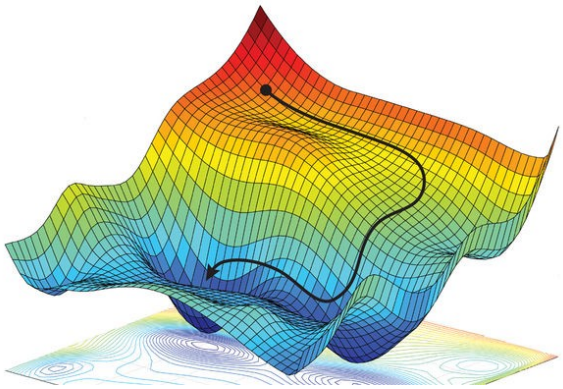


Figure 2.14: Illustration of gradient descent

The algorithm computes in the favorable case the global minimum of f (occurs when f is globally convex) and in the unfavorable case one local minimum of f (occurs when f is not globally convex). In the purpose of avoiding local minimums and speeding convergence, many versions of gradient descent have been introduced.

Note that an iteration over the whole dataset is also called an *epoch*.

2.5.3 Classical Gradient Descent based Algorithms

Here, we will discuss about the *batch gradient descent* , the *stochastic gradient descent* and the *mini-batch stochastic gradient descent* algorithms. The main differ-

ences between them lie in the number of training set examples used to compute the loss and to update the network parameters in each iteration. They will also serve as the reference algorithms for the variant algorithms to be introduced in the following subsections.

2.5.3.1 Bach Gradient Descent

Given the training set $\mathcal{T}$, the batch gradient descent optimization algorithm updates the model parameters with respect to Gradient Descent algorithm introduced previously:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}; \mathcal{T}\right)$$

where :

- $k \geq 1$ denotes the updating iteration. At $k = 0$, vector $\theta^{(0)}$ denotes the initial model parameter vector, which is usually randomly initialized with certain probability distributions (uniform distribution $\mathcal{U}(a, b)$ or normal distribution. $\mathcal{N}(\mu, \sigma^2)$)
- $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}; \mathcal{T}\right)$ is the gradient of the loss with respect to the variable $\boldsymbol{\theta}$. Here the loss is computed on the whole training set $\mathcal{T}$.
- η is the *learning rate* in the gradient descent algorithm. It is a hyper-parameter for the network. A best choice of η speeds convergence (usually very small).

Algorithm 3: Batch Gradient Descent

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, maximum number of epochs N

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
for $ep = 1$ *to* N **do**
    Compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathcal{T})$ on training set $\mathcal{T}$;
    Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathcal{T})$;
return θ

In the batch gradient descent algorithm, to update the model parameters, we need to compute the gradient of the loss function on the total training set in each iteration, which is very time consuming for a large training set. To improve the learning efficiency, stochastic gradient descent and mini-batch gradient descent algorithms have been introduced.

2.5.3.2 Stochastic Gradient Descent

Unlike the *Batch Gradient Descent* which uses the whole training set at each iteration, the *stochastic gradient descent* (often abbreviated SGD) algorithm updates the model parameters by computing the gradient of the loss function instances by instances. Therefore, SGD can be much faster than batch gradient descent, however it may also cause heavy fluctuations towards the optimum in the updating process.

Formally, given the training data $\mathcal{T}$ and the initialized model variable vector $\boldsymbol{\theta}^{(0)}$, at each iteration of SGD, we uniformly sample a pair $(\mathrm{x}^{(i)}; \mathrm{y}^{(i)}) \in \mathcal{T}$, and update of the network parameters is performed through the following equation:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}; \left(\mathrm{x}^{(i)}, \mathrm{y}^{(i)}\right)\right);$$

where here

$$L\left(\boldsymbol{\theta}; \left(\mathrm{x}^{(i)}, \mathrm{y}^{(i)}\right)\right) = \mathcal{L}\left(F\left(\mathrm{x}^{(i)}; \boldsymbol{\theta}\right), \mathrm{y}^{(i)}\right).$$

Algorithm 4: SGD algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, maximum number of epochs N

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
for $ep = 1$ *to* N **do**
  Shuffle the training set $\mathcal{T}$;
  for *each pair* $(\mathrm{x}^{(i)}; \mathrm{y}^{(i)}) \in \mathcal{T}$ **do**
    Compute $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}; (\mathrm{x}^{(i)}; \mathrm{y}^{(i)})\right)$ on the training pair $(\mathrm{x}^{(i)}; \mathrm{y}^{(i)})$;
    Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}; (\mathrm{x}^{(i)}; \mathrm{y}^{(i)})\right)$;
return θ

2.5.3.3 Mini-batch Stochastic Gradient Descent

Mini-batch Stochastic Gradient Descent (or simply called Mini-batch Gradient Descent) is a trade-off between Batch Gradient Descent and SGD. It consists of updating the model parameters with the loss computed on a subset (mini-batch) uniformly sampled from the training set. Formally, for a mini-batch $\mathcal{B} \subset \mathcal{T}$, Mini-batch Gradient

Descent updates the network parameters through the following equation :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}; \mathcal{B}\right);$$

where

$$L(\boldsymbol{\theta}; \mathcal{B}) = \sum_{\left(\mathrm{x}^{(i)}, \mathrm{y}^{(i)}\right) \in \mathcal{B}} \mathcal{L}\left(\hat{\mathrm{y}}^{(i)}, \mathrm{y}^{(i)}\right) = \sum_{\left(\mathrm{x}^{(i)}, \mathrm{y}^{(i)}\right) \in \mathcal{B}} \mathcal{L}\left(F\left(\mathrm{x}^{(i)}; \boldsymbol{\theta}\right), \mathrm{y}^{(i)}\right)$$

.

Algorithm 5: Mini-batch Gradient Descent algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, maximum number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

```
Initialize θ ~ D;
for ep = 1 to N do
    Shuffle the training set T;
    for each mini-batch B ⊂ T do
        Compute ∇_θ L(θ; B);
        Update θ = θ − η · ∇_θ L(θ; B) ;
return θ
```

In the mini-batch gradient descent algorithm, the batch size b is a hyper-parameter. Its values are usually powers of 2 (32, 64, 128 or 256).

Compared with batch gradient descent, the mini-batch gradient descent algorithm is much more efficient especially for the training set of an extremely large size. Meanwhile, compared with the SGD descent, the mini-batch gradient descent algorithm greatly reduces the variance in the model variable updating process and can achieve much more stable convergence (Figure 2.15).

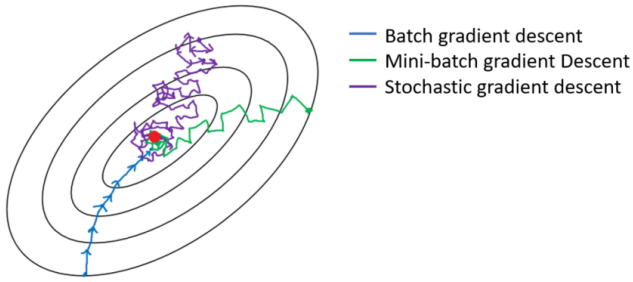


Figure 2.15: Gradient descent based algorithms' trajectory towards minimum

2.5.3.4 Learning Rate Scheduling

Learning Rate Selection: The learning rate η may affect the convergence of the gradient descent algorithms considerably. A choice of a large learning rate may diverge the learning process, while a very small learning rate renders the convergence too slow. Therefore, It is very crucial to select a good learning rate for gradient descent algorithms.

Learning Rate Adjustment : In some situations, a fixed learning rate in the whole training process could not work well. In the initial stages, the algorithm may need a larger learning rate to reach a good optimum fast. However, in the later stages, the algorithm may need to adjust the learning rate with a smaller value to fine-tune the performance instead [14].

2.5.4 Momentum based Algorithms

Momentum based Algorithms are effective gradient based optimization algorithms which smooth the fluctuation encountered in algorithms introduced previously (e.g., Mini batch stochastic gradient descent). They update the network parameters with both the gradient of current iteration as well as the gradients of previous or future iterations simultaneously. In this paper, Momentum based Algorithms will be presented with the Mini-batch Stochastic Gradient Descent as the base learning algorithm.

2.5.4.1 Momentum

Momentum optimizes the model parameters with the following equation :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \Delta \mathbf{v}^{(k)}, \text{ where } \Delta \mathbf{v}^{(k)} = \rho \cdot \Delta \mathbf{v}^{(k-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}\right) \quad (2.7)$$

- $L(\boldsymbol{\theta}) = L(\boldsymbol{\theta}; \mathcal{B})$ is the loss term computed on a mini-batch $\mathcal{B}$.
- $\rho \in [0; 1]$ denotes the weight of the momentum term.

Lemma 1. *$\Delta \mathbf{v}^{(k)}$ can be expanded with the following equation :*

$$\Delta \mathbf{v}^{(\tau)} = \sum_{t=0}^{k-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1-t)}\right)$$

Proof. By induction
1. Initialization : if $k = 1$ we have :

$$\begin{aligned} \Delta \mathbf{v}^{(1)} &= \rho \cdot \Delta \mathbf{v}^{(0)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(0)}\right) \\ &= (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(0)}\right) \end{aligned}$$

where $\Delta \mathbf{v}^{(0)}$ is initialized as a zero vector.

2. Assumption : Let's assume for $k = q$ that:

$$\Delta \mathbf{v}^{(q)} = \sum_{t=0}^{q-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q-1-t)}\right)$$

3. Induction : For $k = q + 1$, according to equation 3.1, $\Delta \mathbf{v}^{(q+1)}$ can be formulated as :

$$\begin{aligned} \Delta \mathbf{v}^{(q+1)} &= \rho \cdot \Delta \mathbf{v}^{(q)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q)}\right) \\ &= \rho \cdot \sum_{t=0}^{q-1} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q-1-t)}\right) + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q)}\right) \\ &= \sum_{t=1}^{p} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q-t)}\right) + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q)}\right) \\ &= \sum_{t=0}^{q} \rho^t \cdot (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(q-t)}\right) \end{aligned}$$

which ends the proof. ■

The Momentum algorithm can achieve a faster convergence than classical gradient descent based algorithms. Since $\rho \in [0; 1]$ the gradients of the iterations which are far away from the current iteration will decay exponentially. Momentum based algorithms are also called *Exponentially weighted average algorithms*.

Algorithm 6: Momentum algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, Momentum weight parameter ρ, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

```
Initialize θ ~ D;
Initialize Momentum term Δv = 0;
for ep = 1 to N do
    Shuffle the training set T;
    for each mini-batch B ⊂ T do
        Compute ∇_θ L(θ; B);
        Update term Δv = ρ · Δv + (1 − ρ) · ∇_θ L(θ; B);
        Update θ = θ − η · Δv ;
return θ
```

2.5.4.2 Nesterov Accelerated Gradient

Unlike the previous gradient descent based algorithms which all update the variables based on the gradients at both the past or current points without knowledge about the future, the *Nesterov Accelerated Gradient (NAG) [17]* algorithm updates the parameters with the gradient at an approximated future point instead [14]. This process allows to update the variables much more effectively.

At iteration K, the variable $\boldsymbol{\theta}^{(k)}$ can be estimated by $\hat{\boldsymbol{\theta}}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \rho \cdot \Delta\mathbf{v}^{(k-1)}$ where $\Delta\mathbf{v}^{(k-1)}$ denotes the momentum term at iteration $k-1$. Formally, NAG optimizes the model parameters through the following equation :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \Delta\mathbf{v}^{(k)},$$

where

$$\begin{cases} \hat{\boldsymbol{\theta}}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \eta \cdot \rho \cdot \Delta\mathbf{v}^{(k-1)} \\ \Delta\mathbf{v}^{(k)} = \rho \cdot \Delta\mathbf{v}^{(k-1)} + (1-\rho) \cdot \nabla_{\boldsymbol{\theta}} L\left(\hat{\boldsymbol{\theta}}^{(k)}\right) \end{cases}$$

Algorithm 7: Nesterov Accelerated Gradient algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, Momentum weight parameter ρ, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
Initialize Momentum term $\Delta \mathbf{v} = 0$;
for $ep = 1$ *to* N **do**
  Shuffle the training set $\mathcal{T}$;
  for *each mini-batch* $\mathcal{B} \subset \mathcal{T}$ **do**
    Compute $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta} - \eta \cdot \rho \cdot \Delta \mathbf{v}$;
    Compute $\nabla_{\boldsymbol{\theta}} L\left(\hat{\boldsymbol{\theta}}; \mathcal{B}\right)$;
    Update term $\Delta \mathbf{v} = \rho \cdot \Delta \mathbf{v} + (1 - \rho) \cdot \nabla_{\boldsymbol{\theta}} L(\hat{\boldsymbol{\theta}}; \mathcal{B})$;
    Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \cdot \Delta \mathbf{v}$;
return θ

2.5.5 Adaptive Gradient based Learning Algorithms

Adaptive Gradient based Learning Algorithms are gradient descent based algorithms which update the model parameters with adaptive learning rates.

2.5.5.1 Adagrad

The learning rate in the optimization algorithms introduced previously are fixed and identical for all the components of vector $\boldsymbol{\theta}$. However, in the variable updating process, different components may require different learning rates. Indeed, for the variable components reaching the optimum, a smaller learning rate is needed; while for the components far away from the optimum, a larger learning rate may be required in order to reach the optimum faster.

Adagrad [18] was introduced to solve these problems. It updates the model parameters through the following equation :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \frac{\eta}{\sqrt{\operatorname{diag}\left(\mathbf{G}^{(k)}\right) + \epsilon \cdot \mathbf{I}}} \mathbf{g}^{(k-1)} \text{ where } \begin{cases} \mathbf{g}^{(k-1)} & = \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}\right) \\ \mathbf{G}^{(k)} & = \sum_{t=0}^{k-1} \mathbf{g}^{(t)} \left(\mathbf{g}^{(t)}\right)^{\top} \end{cases}$$

- $diag(\mathbf{G})$ denotes a diagonal matrix of the same dimensions as $\mathbf{G}$ and whose diagonal elements are the same as $\mathbf{G}$.
- ϵ is a smoothing term whose role is to avoid diving by zero.
- Matrix $\mathbf{G}^{(k)}$ keeps records of the computed past gradients from the beginning until the current iteration. Since the values of the diagonal of $\mathbf{G}^{(k)}$ are different, the learning rates of the components of the parameter θ are also different, e.g., the learning rate for $\boldsymbol{\theta}_i$ at iteration τ will be $\eta_i^{(k)} = \frac{\eta}{\sqrt{\mathbf{G}_{i,i}^{(k)}+\epsilon}}$. Moreover, because the matrix $\mathbf{G}^{(k)}$ will be different at each iteration, the learning rate for the same component in different iterations will also be different.

To be more explicit, if the model parameter $\boldsymbol{\theta} \in \mathbb{R}^m$, Adagrad can be expanded into this form :

$$\begin{bmatrix} \boldsymbol{\theta}_1^{(k)} \\ \boldsymbol{\theta}_2^{(k)} \\ \vdots \\ \boldsymbol{\theta}_m^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta}_1^{(k-1)} \\ \boldsymbol{\theta}_2^{(k-1)} \\ \vdots \\ \boldsymbol{\theta}_m^{(k-1)} \end{bmatrix} - \eta \left(\begin{bmatrix} \epsilon & 0 & \cdots & 0 \\ 0 & \epsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \epsilon \end{bmatrix} + \begin{bmatrix} G_{1,1}^{(k)} & 0 & \cdots & 0 \\ 0 & G_{2,2}^{(k)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & G_{m,m}^{(k)} \end{bmatrix} \right)^{-1/2} \cdot \begin{bmatrix} g_1^{(k-1)} \\ g_2^{(k-1)} \\ \vdots \\ g_m^{(k-1)} \end{bmatrix}$$

which can be simplified as :

$$\begin{bmatrix} \boldsymbol{\theta}_1^{(k)} \\ \boldsymbol{\theta}_2^{(k)} \\ \vdots \\ \boldsymbol{\theta}_m^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta}_1^{(k-1)} \\ \boldsymbol{\theta}_2^{(k-1)} \\ \vdots \\ \boldsymbol{\theta}_m^{(k-1)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\epsilon+G_{1,1}^{(k)}}} g_1^{(k-1)} \\ \frac{\eta}{\sqrt{\epsilon+G_{2,2}^{(k)}}} g_2^{(k-1)} \\ \vdots \\ \frac{\eta}{\sqrt{\epsilon+G_{m,m}^{(k)}}} g_m^{(k-1)} \end{bmatrix}$$

Algorithm 8: Adagrad algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
Initialize matrix $\mathbf{G} = 0$;
for $ep = 1$ *to* N **do**
    Shuffle the training set $\mathcal{T}$;
    for *each mini-batch* $\mathcal{B} \subset \mathcal{T}$ **do**
        Compute vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathcal{B})$ on $\mathcal{B}$;
        Update matrix $\mathbf{G} = \mathbf{G} + \mathbf{g}\mathbf{g}^T$;
        Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\mathrm{diag}(\mathbf{G}) + \epsilon \cdot \mathbf{I}}} \mathbf{g}$;
return θ

As the iteration continues, the values on the diagonal of matrix $\mathbf{G}$ will be increasing, and therefore the learning rates of the components of the model variable will decrease monotonically in the learning process. It will thus create problems for the learning in later iterations, since the variables can no longer be effectively updated with information from the training data.

2.5.5.2 RMSprop

RMsprop [19] is an advanced version of *Adagrad* and was introduced in the purpose of solving the monotonically decreasing learning rate problem encountered with *Adagrad*. The core idea of *RMSprop* is to decrease the weight of the past accumulated gradients in the matrix $\mathbf{G}$ defined in Adagrad, and also to allow the adjustment of learning rate in the learning process. It updates the model variable with the following equation:

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \frac{\eta}{\sqrt{\mathrm{diag}\left(\mathbf{G}^{(k)}\right) + \epsilon \cdot \mathbf{I}}} \mathbf{g}^{(k-1)},$$

where

$$\begin{cases} \mathbf{g}^{(k-1)} = \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}\right) \\ \mathbf{G}^{(k)} = \rho \cdot \mathbf{G}^{(k-1)} + (1-\rho) \cdot \mathbf{g}^{(k-1)} \left(\mathbf{g}^{(k-1)}\right)^{\top} \end{cases}$$

Parameter ρ is the weight of the past accumulated gradients is usually set as 0.9 according to [19]. RMSprop algorithm can be compressed as :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \frac{\eta}{RMS\left(\mathbf{g}^{(k-1)}\right)}\mathbf{g}^{(k-1)} \text{ where } RMS(\mathbf{g}^{(k-1)}) = \sqrt{\operatorname{diag}\left(\mathbf{G}^{(k)}\right) + \epsilon \cdot \mathbf{I}}$$

RMS denotes root mean square metric on vector $\mathbf{g}^{(k-1)}$.

Algorithm 9: RMSprop algorithm

Input: Training set $\mathcal{T}$, learning rate η, Initialization distribution $\mathcal{D}$, Parameter ρ, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L\left(\boldsymbol{\theta}; \mathcal{T}\right)$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
Initialize matrix $\mathbf{G} = 0$;
for $ep = 1$ *to* N **do**
    Shuffle the training set $\mathcal{T}$;
    for *each mini-batch* $\mathcal{B} \subset \mathcal{T}$ **do**
        Compute vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}; \mathcal{B}\right)$ on $\mathcal{B}$;
        Update matrix $\mathbf{G} = \rho \cdot \mathbf{G} + (1-\rho) \cdot \mathbf{g}\mathbf{g}^T$;
        Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\operatorname{diag}(\mathbf{G}) + \epsilon \cdot \mathbf{I}}}\mathbf{g}$;
return θ

2.2.5.3 Adadelta

Like RMSprop, *Adadelta* [20] was introduced to address the monotonically decreasing learning rate problem in Adagrad. Adadelta was performed to eliminate the learning rate from the updating equations of the model parameters. Furthermore, unlike the other optimization algorithms which do not take into account the homogeneity of the units (*km*, *kg*, *s*, etc) of the variable, Adadelta addresses this problem by looking at the second-order methods (*Newton method*).

If the parameter $\boldsymbol{\theta}$ has a specific units, then the updates in parameter, i.e $\Delta\boldsymbol{\theta} = \Delta_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ should have the same units as well. Such condition is not satisfied in the algorithms introduced previously. For example, in SGD :

$$\text{units of } \Delta\boldsymbol{\theta} \propto \text{ units of } \mathbf{g} \propto \text{ units of } \frac{\partial L(\cdot)}{\partial \boldsymbol{\theta}} \propto \frac{1}{\text{units of } \boldsymbol{\theta}}.$$

Adadelta solves such problem with Newton's method that uses Hessian approximation.

In Newton's method, we have

$$\text{units of } \Delta\boldsymbol{\theta} \propto \text{ units of } \mathbf{H}^{-1}\mathbf{g} \propto \text{ units of } \frac{\frac{\partial L(\cdot)}{\partial \boldsymbol{\theta}}}{\frac{\partial^2 L(\cdot)}{\partial \boldsymbol{\theta}^2}} \propto \text{ units of } \boldsymbol{\theta}$$

where $\mathbf{H} = \frac{\partial^2 L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}$ is the Hessian matrix.

In Adaleta, the updating term is formulated as $\Delta\boldsymbol{\theta} = -\frac{\frac{\partial L(\cdot)}{\partial \boldsymbol{\theta}}}{\frac{\partial^2 L(\cdot)}{\partial \boldsymbol{\theta}^2}}$ that implies that

$$\mathbf{H}^{-1} = -\frac{1}{\frac{\partial^2 L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2}} = -\frac{\Delta\boldsymbol{\theta}}{\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}}.$$

Formally, Adadelta proposes to udpate the model parameters with the following equation :

$$\begin{aligned}\boldsymbol{\theta}^{(k)} &= \boldsymbol{\theta}^{(k-1)} - \left(\mathbf{H}^{(k)}\right)^{-1}\mathbf{g}^{(k-1)} \\ &= \boldsymbol{\theta}^{(k-1)} - \frac{\Delta\boldsymbol{\theta}^{(k)}}{\frac{\partial L\left(\boldsymbol{\theta}^{(k)}\right)}{\partial \boldsymbol{\theta}}}\mathbf{g}^{(k-1)}\end{aligned}$$

In the above equation, the term $\Delta\boldsymbol{\theta}^{(k)}$ in the current iteration is unknown yet so Adadelta proposes to approximate it by replacing $\Delta\boldsymbol{\theta}$ by $RMS(\Delta\boldsymbol{\theta})$. Therefore, the parameters updating equation in Adadelta can be formally written as :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \frac{RMS\left(\Delta\boldsymbol{\theta}^{(k-1)}\right)}{RMS\left(\mathbf{g}^{(k-1)}\right)}\mathbf{g}^{(k-1)}$$

Algorithm 10: Adadelta algorithm

Input: Training set $\mathcal{T}$, Initialization distribution $\mathcal{D}$, Parameter ρ, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
Initialize matrix $\mathbf{G} = 0$;
Initialize matrix $\Theta = 0$;
for $ep = 1$ *to* N **do**
    Shuffle the training set $\mathcal{T}$;
    for *each mini-batch* $\mathcal{B} \subset \mathcal{T}$ **do**
        Compute vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathcal{B})$ on $\mathcal{B}$;
        Update matrix $\mathbf{G} = \rho \cdot \mathbf{G} + (1-\rho) \cdot \mathbf{g}\mathbf{g}^T$;
        Computer updating vector $\Delta\boldsymbol{\theta} = -\frac{\sqrt{\mathrm{diag}(\boldsymbol{\Theta}) + \epsilon \cdot \mathbf{I}}}{\sqrt{\mathrm{diag}(\mathbf{G}) + \epsilon \cdot \mathbf{I}}}\mathbf{g}$;
        Update $\Theta = \rho \cdot \Theta + (1-\rho) \cdot \Delta\boldsymbol{\theta}(\Delta\boldsymbol{\theta})^\top$;
        Update $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$;
return θ

2.5.6 Adam

Adaptive Moment Estimation (Adam) [21] use both the ideas behind the Momentum algorithms and the algorithms with adaptive learning rate. More specifically, Adam is a combination of the RMSprop algorithm and the Momentum algorithm. Adam records both of the historical squared first-order gradients and the historical first-order gradients and both of them will decay exponentially. Let's denote by $\mathbf{m}^{(\mathbf{k})}$ and $\mathbf{v}^{(\mathbf{k})}$ the first-order gradients and the squared first-order gradients respectively.

$$\mathbf{m}^{(k)} = \beta_1 \cdot \mathbf{m}^{(k-1)} + (1-\beta_1) \cdot \mathbf{g}^{(k-1)}$$
$$\mathbf{v}^{(k)} = \beta_2 \cdot \mathbf{v}^{(k-1)} + (1-\beta_2) \cdot \mathbf{g}^{(k-1)} \odot \mathbf{g}^{(k-1)}$$

- $\mathbf{g}^{(k-1)} = \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{(k-1)}\right)$
- $\mathbf{g}^{(k-1)} \odot \mathbf{g}^{(k-1)}$ denotes the **Hadamard product** or **element-wise product** of the vectors.

In the above equation, because $\mathbf{m}^{(\mathbf{0})}$ and $\mathbf{v}^{(\mathbf{0})}$ are initialized with zero values, $\mathbf{m}^{(\mathbf{k})}$ and

$\mathbf{v^{(k)}}$ are biased toward zero and especially when β_1 and β_2 are close to 1 [14, 21]. To address this problem, $\mathbf{m^{(k)}}$ and $\mathbf{v^{(k)}}$ are re-scaled as following :

$$\hat{\mathbf{m}}^{(k)} = \frac{\mathbf{m}^{(k)}}{1-\beta_1^k}$$
$$\hat{\mathbf{v}}^{(k)} = \frac{\mathbf{v}^{(k)}}{1-\beta_2^k}$$

β_1^k and β_2^k denote the power of β_1 and β_2 with iteration k respectively.

Formally, Adam optimizes the model parameters through the following equation :

$$\boldsymbol{\theta}^{(k)} = \boldsymbol{\theta}^{(k-1)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(k)}}+\epsilon} \odot \hat{\mathbf{m}}^{(k)}$$

Algorithm 11: Adadelta algorithm

Input: Training set $\mathcal{T}$,Learning rate η , Initialization distribution $\mathcal{D}$, Decay Parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, total number of epochs N, batch size b

Output: Parameter $\boldsymbol{\theta}$ which Globally or locally minimizes the loss function $L(\boldsymbol{\theta}; \mathcal{T})$

Initialize $\boldsymbol{\theta} \sim \mathcal{D}$;
Initialize vector $\mathbf{v} = 0$;
Initialize vector $\mathbf{m} = 0$;
Initialize step $k = 0$;
for $ep = 1$ *to* N **do**
    Shuffle the training set $\mathcal{T}$;
    for *each mini-batch* $\mathcal{B} \subset \mathcal{T}$ **do**
        Update step $k = k + 1$ Compute vector $\mathbf{g} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}; \mathcal{B})$ on $\mathcal{B}$;
        Update vector $\mathbf{m} = \beta_1 \cdot \mathbf{m} + (1-\beta_1) \cdot \mathbf{g}$;
        Update vector $\mathbf{v} = \beta_2 \cdot \mathbf{v} + (1-\beta_2) \cdot \mathbf{g} \odot \mathbf{g}$;
        Re-scale vector $\hat{\mathbf{m}} = \mathbf{m} / \left(1-\beta_1^k\right)$;
        Re-scale vector $\hat{\mathbf{v}} = \mathbf{v} / \left(1-\beta_2^k\right)$;
        Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}}+\epsilon} \odot \hat{\mathbf{m}}$;
return $\boldsymbol{\theta}$

Chapter 3

STATE OF THE ART DEEP LEARNING METHODS FOR OBJECT DETECTION

Computer vision is a branch of artificial intelligence and refers to the various techniques that allow computers to see and understand the content of images and videos. Thanks to CNN, computer vision has known a boom and one is able to perform many computer vision tasks including *image classification*, *object localization*, *object detection* and *object segmentation*.

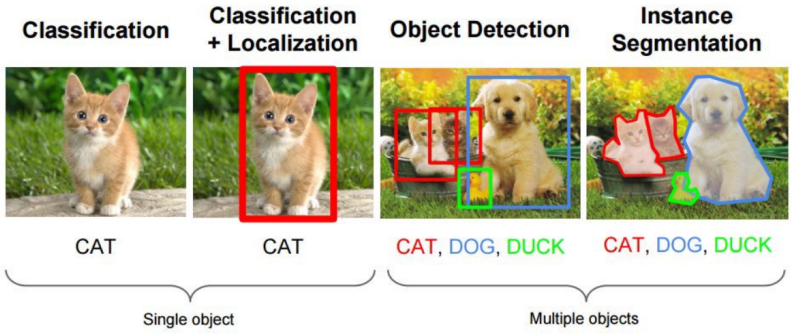


Figure 3.1: Computer vision tasks

Image classification refers to the task of identifying the class an object in an image while in *object localization*, we classify and locate an object of interest via a bounding box. *Object detection* consists of identifying multiple objects and localizing them through bounding boxes. As for *segmentation*, it refers to identifying class categories of the pixels in the image e.g which objects the pixels in the image belongs to.

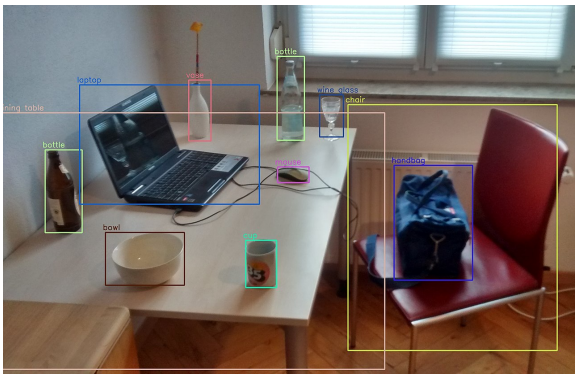
These techniques have found many industrial applications ranging from self-driving

cars to facial recognition, medical sciences, robotics and supervision systems. In this work, we are only interested in object detection techniques and therefore, in this chapter, we will study state of the art deep learning methods for object detection.

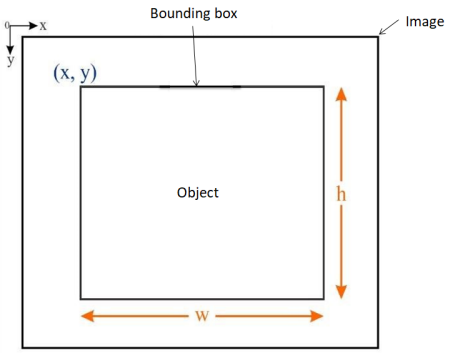
3.1 Introduction

3.1.1 What is object detection in computer vision?

Object detection is a field of computer vision that refers to the task of detecting and localizing all the instances of objects of interest (e.g. cats, dogs, humans, furniture, etc.) in digital images and videos. Usually, the localization is provided as a bounding box containing the object (Figure 3.2). Object detection is capable to provide valuable information for semantic understanding of images and videos, and has many applications, such as precise localization of diseases from medical images, human behavior analysis, face recognition and autonomous driving [2].



(a) Objects detected with a deep learning model.



(b) Object localization by bounding box.

Figure 3.2: Object detection in an image

3.1.2 Performance evaluation of a detection system

Here we introduce the most popular metrics used to evaluate object detection models. This subsection is very inspired from [23].

3.1.2.1 Important definitions

Intersection Over Union (IOU) *IOU* measures the overlap between two bounding boxes. The IOU score (belongs to the interval $[0, 1]$) is used to know if a detection is valid (True Positive) or not (False Positive). Given a ground truth bounding box B_{gt} and predicted bounding box B_p, *IOU* is defined as :

$$\text{IOU} = \frac{\text{area}\left(B_p \cap B_{gt}\right)}{\text{area}\left(B_p \cup B_{gt}\right)}$$

$$IOU = \frac{\text{area of overlap}}{\text{area of union}} =$$

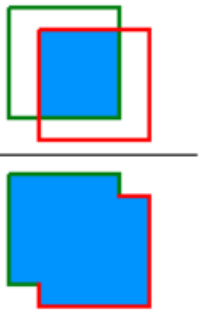


Figure 3.3: IOU between a ground truth bounding box (in green) and a detected bounding box (in red)

True Positive (TP) A correct detection. A detection with IOU $\geq$ *threshold*.

False Positive (FP) A wrong detection. A detection with IOU $<$ *threshold*.

False Negative (FN) A ground truth not detected.

True Negative (TN) A corrected miss detection. Not used in object detection tasks.

The *threshold* value is chosen by the user and is usually set to 0.5, 0.75 or 0.95. In the cases where there are many detections for same object (more than one detections with IOU$\geq$ *threshold*), the detection with the highest IOU is considered **TP** and the others are considered **FP**.

Precision *Precision* is the fraction of relevant instances among the retrieved instances. It evaluates the capacity of the model to identify **only** the relevant objects.

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{TP}{\text{all detections}}$$

Recall *Recall* or *sensitivity* is the fraction of the total number of relevant instances that were actually retrieved. It measures the ability of the model to identify **all** the relevant cases (the ability to detect all ground truth bounding boxes).

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{TP}{\text{all ground truths}}$$

3.1.2.2 Metrics used to evaluate object detection algorithms

Precision $\times$ Recall curve The Precision $\times$ Recall curve is a good metric to evaluate the performance of an object detection algorithm. An object detector of a certain class is considered good if its precision remains high when its recall increases, i.e if you tune the confidence threshold, the precision and recall will still be high. In other words a perfect detector detect only relevant objects (zero False Positives = high precision) while finding all ground truth objects (zero False Negatives = high recall). A poor object detector needs to increase the number of detected objects (increasing False Positives = lower precision) in order to retrieve all ground truth objects (high recall).

Average Precision Comparing different detectors with the Precision $\times$ Recall curve is equivalent to comparing different curves in the same plot. However, as we know, comparing different curves in the same plot usually is not an easy task. Therefore, the *Average Precision (AP)* metric has been introduced to evaluate the performance of object detectors by calculating the area under the curve (AUC) of the Precision $\times$ Recall curve:

$$\text{AP} = \int_0^1 \rho(r)dr \tag{3.1}$$

Since precision and recall are always between 0 and 1, AP falls in the interval $[\,0\,;\,1\,]$. There exists two popular numerical methods [25] used to approximate the integral defined by equation (3.1):

- **11-point interpolation**
  The 11-point interpolation is an interpolation method that computes the AUC of

the Precision $\times$ Recall curve by averaging the precision at a set of eleven equally spaced recall points $\{0, 0.1, 0.2, ..., 1\}$:

$$\mathrm{AP} = \frac{1}{11} \sum_{r \in \{0, 0.1, ..., 1\}} \rho_{\text{interp}}\,(r)$$

with $\rho_{\text{interp}}\,(r) = \max_{\tilde{r}:\tilde{r} \geq r} \rho(\tilde{r})$ where $\rho(\tilde{r})$ is the measured precision at recall $\tilde{r}$.

The AP is computed by interpolating the precision only at the 11 levels r taking the maximum precision whose recall value is greater than r.

- **Interpolating all points**
  Instead of using only 11 equally spaced points, we could interpolate through all points:

$$\sum_{n \geq 0} (r_{n+1} - r_n)\, \rho_{\text{interp}}\ (r_{n+1})$$

  with $\rho_{\text{interp}}\,(r_{n+1}) = \max_{\tilde{r}:\tilde{r} \geq r_{n+1}} \rho(\tilde{r})$ where $\rho(\tilde{r})$ is the measured precision at recall $\tilde{r}$.

Illustration example Since it is not easy to understand these metrics, we are going to illustrate them with an example. In the Figure 3.4, there 7 images, 15 ground truth objects (the green bounding boxes) and 24 detected objects (the red bounding boxes). Each detected object has a confidence probability and is identified by a letter (A,B,...,Y). The table in Figure 3.5 tells us if a detection is TP or FP. The threshold is set to 0.3. In some images (2, 3, 4, 5, 6 and 7), there are more than one detection with IOU overlap with a ground truth greater than 0.3. In those cases, the detection with the highest IOU is considered as TP and the others FP.

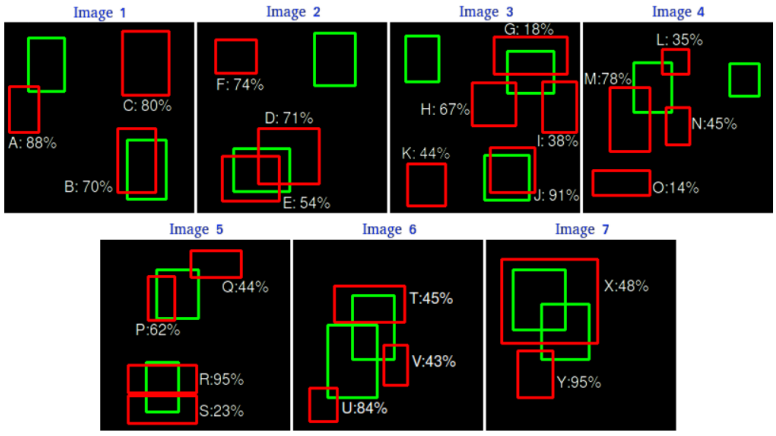


Figure 3.4: Illustration example

Images	Detections	Confidences	TP or FP
Image 1	A	88%	FP
Image 1	B	70%	TP
Image 1	C	80%	FP
Image 2	D	71%	FP
Image 2	E	54%	TP
Image 2	F	74%	FP
Image 3	G	18%	TP
Image 3	H	67%	FP
Image 3	I	38%	FP
Image 3	J	91%	TP
Image 3	K	44%	FP
Image 4	L	35%	FP
Image 4	M	78%	FP
Image 4	N	45%	FP
Image 4	O	14%	FP
Image 5	P	62%	TP
Image 5	Q	44%	FP
Image 5	R	95%	TP
Image 5	S	23%	FP
Image 6	T	45%	FP
Image 6	U	84%	FP
Image 6	V	43%	FP
Image 7	X	48%	TP
Image 7	Y	95%	FP

Figure 3.5: Detections as TP or FP

To plot the precision $\times$ recall curve, the detections are first ordered by confidences and then one computes the precision and recall values of the accumulated TP or FP detections (Figure 3.6).

Images	Detections	Confidences	TP	FP	Acc TP	Acc FP	Precision	Recall
Image 5	R	95%	1	0	1	0	1	0.0666
Image 7	Y	95%	0	1	1	1	0.5	0.0666
Image 3	J	91%	1	0	2	1	0.6666	0.1333
Image 1	A	88%	0	1	2	2	0.5	0.1333
Image 6	U	84%	0	1	2	3	0.4	0.1333
Image 1	C	80%	0	1	2	4	0.3333	0.1333
Image 4	M	78%	0	1	2	5	0.2857	0.1333
Image 2	F	74%	0	1	2	6	0.25	0.1333
Image 2	D	71%	0	1	2	7	0.2222	0.1333
Image 1	B	70%	1	0	3	7	0.3	0.2
Image 3	H	67%	0	1	3	8	0.2727	0.2
Image 5	P	62%	1	0	4	8	0.3333	0.2666
Image 2	E	54%	1	0	5	8	0.3846	0.3333
Image 7	X	48%	1	0	6	8	0.4285	0.4
Image 4	N	45%	0	1	6	9	0.4	0.4
Image 6	T	45%	0	1	6	10	0.375	0.4
Image 3	K	44%	0	1	6	11	0.3529	0.4
Image 5	Q	44%	0	1	6	12	0.3333	0.4
Image 6	V	43%	0	1	6	13	0.3157	0.4
Image 3	I	38%	0	1	6	14	0.3	0.4
Image 4	L	35%	0	1	6	15	0.2857	0.4
Image 5	S	23%	0	1	6	16	0.2727	0.4
Image 3	G	18%	1	0	7	16	0.3043	0.4666
Image 4	O	14%	0	1	7	17	0.2916	0.4666

Figure 3.6: Detections are firstly odered by confidence and then accumulated precision and recall values are calculated.

We can now plot the precision $\times$ recall curve.

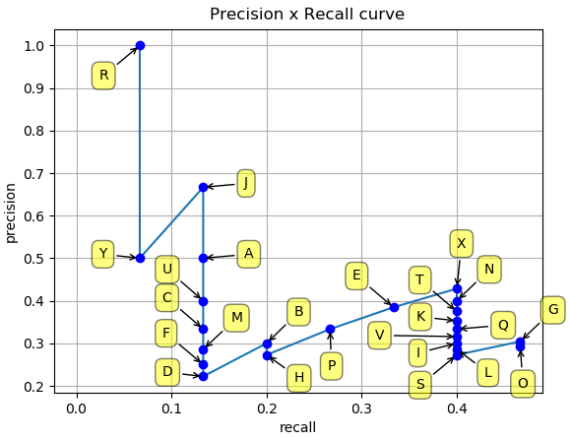


Figure 3.7: Precision vs Recall curve

Let's compute the AP score with the *11-point interpolation* and *Interpolating all points* methods.

- **11-point interpolation**

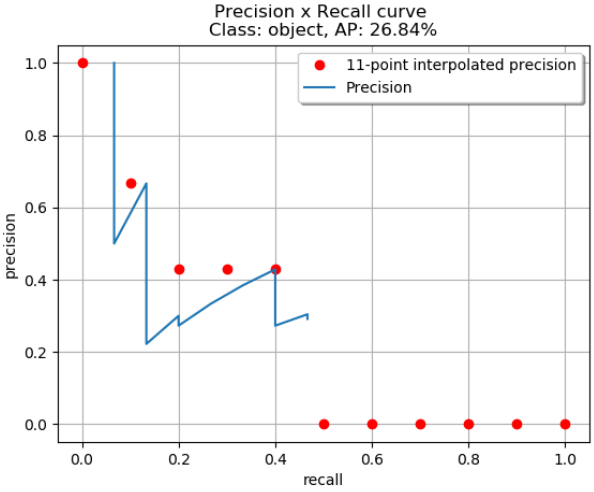
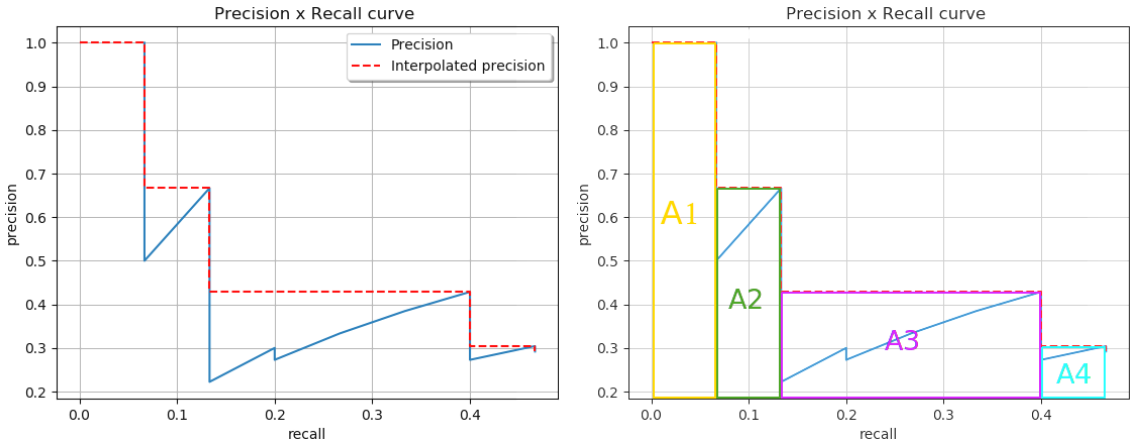


Figure 3.8: Example: computing the AP with the 11-point interpolation.

$$AP = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 1\}} \rho_{\text{interp}}(r)$$
$$AP = \frac{1}{11}(1 + 0.6666 + 0.4285 + 0.4285 + 0.4285 + 0 + 0 + 0 + 0 + 0 + 0)$$
$$AP = 0.2684 = 26.84\%$$

- **Interpolating all points**



(a) Interpolated precision with all points.

(b) we can divide the AUC into 4 areas (A1, A2, A3 and A4).

Figure 3.9: Example: computing the AP with all-point interpolation.

$$AP = A1 + A2 + A3 + A4$$

with:

$$A1 = (0.0666 - 0) \times 1 = 0.0666$$
$$A2 = (0.1333 - 0.0666) \times 0.6666 = 0.04446222$$
$$A3 = (0.4 - 0.1333) \times 0.4285 = 0.11428095$$
$$A4 = (0.4666 - 0.4) \times 0.3043 = 0.02026638$$
$$AP = 0.0666 + 0.04446222 + 0.11428095 + 0.02026638$$
$$AP = 0.24560955$$
$$AP = 0.2456 = 24.56\%$$

3.1.3 Overview of modern object detectors

Deep learning algorithms for object detection can be divided into two groups, two-stage, and one-stage detection models. Two-stage detectors split the process into two tasks. In the first stage, regions of interest (ROI) are proposed by either a network (*Region Proposal Network*) or a classical image processing algorithm (*Selective Search*), and in the final stage, bounding box regression and classification is being performed inside the proposed regions. One-stage detectors predict the bounding boxes and their classes at once. Usually, two-stage models are more precise in terms of localization and classification accuracy, but in terms of processing are slower then one-stage detectors. Both of these detectors contain two stage networks: a backbone network for feature extraction and head networks for classification and regression. Most of the cases, the backbone is some classical CNN such as VGG-16, ResNet, etc. pre-trained on ImageNet [37].

3.1.4 Anchor boxes

Some of the methods we're going to describe use the concept of *anchor boxes*. So before going further, let's take a look at this concept of anchor boxes. Anchor boxes are a set of prior bounding boxes of a certain height and width (Figure 3.10). The purpose of using anchor boxes is to make easier for the network to learn the scale and aspect ratio of specific object classes. The network predicts offsets and confidences for anchor boxes instead of directly predicting bounding boxes coordinates. [40] showed the advantages of using anchor boxes: their use allows the detector to detect multiple objects, objects of different scales, and overlapping objects. Anchor boxes are generated by running *k-means* algorithm on the training set bounding boxes. Here, the Euclidean distance used in classical *k-means* is replaced with a novel distance metric:

$$d(\text{ box, centroid }) = 1 - \text{IOU}(\text{box, centroid})$$

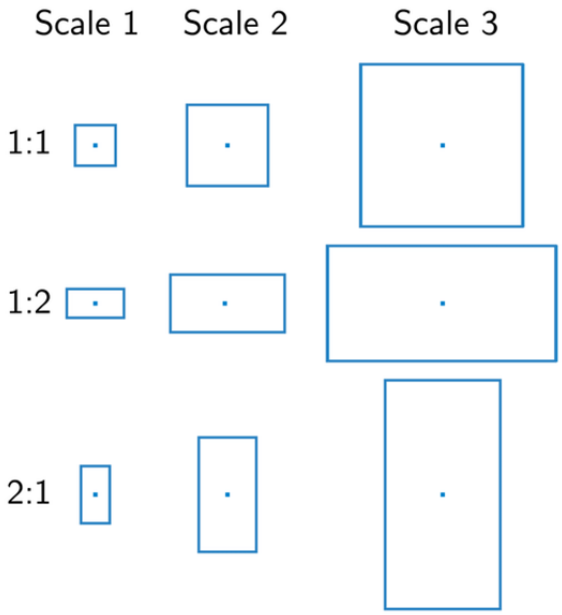


Figure 3.10: 9 anchors boxes with 3 different scales and 3 different aspect ratios.

3.2 One-stage detectors

The most popular one-stage detectors are YOLO [33], SSD [34] and RetinaNet [35]. In this work, we only describe YOLO.

3.2.1 YOLO

3.2.1.1 Basic functioning and architecture of the network

YOLO (*You Only Look Once*) [33] is a detection system targeted for speed (real time). It performs detections in a single forward-pass. The model backbone, inspired by GoogleNet [36] has 24 convolutional layers followed by 2 fully connected layers (Figure 3.11).

With YOLO, the input image is divided into a two dimensional grid of shape $S \times S$ (Figure 3.12) and for each cell of the grid, B bounding boxes and confidence scores for those boxes are predicted. The confidence score measures how much is the cell confident that it contains an object. Formally, the confidence score is defined as $\Pr(\text{ Object }) * \text{IOU}_{\text{pred}}^{\text{retuth}}$: if there is no object in the cell, the confidence score should be zero; otherwise, the confidence score should be equal to the IOU (intersection

over union) between the predicted bounding box and the ground truth. The cell that contains the center of an object is responsible for the detection of that object.

In each grid cell, C conditional probabilities $\Pr\left(\text{ Class }_i|\text{ Object }\right)$ are also predicted. These probabilities are predicted only if the cell contains an object.

Each bounding box is encoded by 5 numbers: $\mathrm{x}, \mathrm{y}, \mathrm{w}, \mathrm{h}$ and the confidence score where the (x, y) coordinates denote the center of the box relative to the bounds of the grid cell and w, h its width and height.

Finally, the output of the network is a matrix of shape $S \times S \times (B \times 5 + C)$.

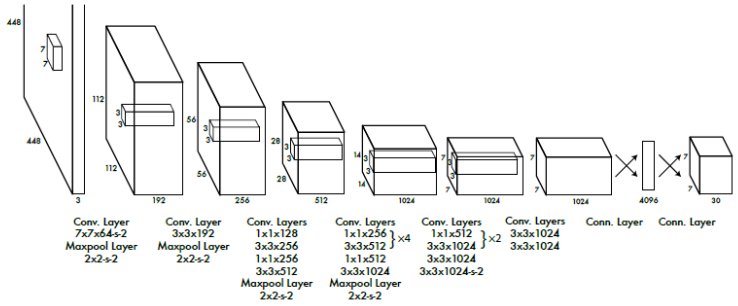


Figure 3.11: YOLO's architecture: 24 convolutional layers followed by 2 fully connected layers.

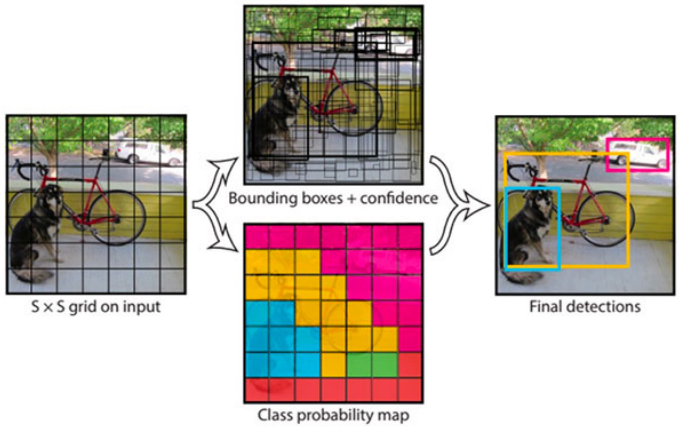


Figure 3.12: YOLO as regression problem: the input image is divided into a grid and bounding boxes, confidence in those boxes and class probabilities are simultaneously predicted for each cell in the grid.

3.2.1.2 Training

YOLO is trained using transfer learning. Explicitly, YOLO is firstly pretrained on Imagenet classification dataset and then fine-tuned to perform detection.

During training, the following loss function is attempted to be minimized:

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(\mathrm{x}_i - \hat{\mathrm{x}}_i)^2 + (\mathrm{y}_i - \hat{\mathrm{y}}_i)^2 \right] \\
&+\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{\mathrm{w}_i} - \sqrt{\hat{\mathrm{w}}_i})^2 + (\sqrt{\mathrm{h}_i} - \sqrt{\hat{\mathrm{h}}_i})^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{ classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2
\end{aligned}
\tag{3.2}
$$

where :

- In the cell i, $(\mathrm{x}_i\ ,\ \mathrm{y}_i)$ denote the center of the box relative to the bounds of the grid cell, $(\mathrm{w}_i\ ,\ \mathrm{h}_i)$ are the normalized width and height relative to the image size, C_i is the confidence score, $\mathbb{1}_i^{\text{obj}}$ indicates if there is an object in the cell i, $\mathbb{1}_{ij}^{\text{obj}}$ indicates that the jth bounding box predictor in the cell i is responsible for that prediction.

- The loss function penalizes the classification error only if there is an object in the cell. It also only penalizes bounding box coordinate error if that bounding box is responsible for the ground truth box (i.e. has the highest IOU of any predictor in that grid cell).

- Parameters λ_{coord} and λ_{noobj} are used to increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don't contain objects. Usually, they are fixed to $\lambda_{\text{coord}} = 5$ and $\lambda_{\text{noobj}} = 0.5$.

Dropout [38] and data augmentation (random scaling, translations, adjust and saturation) are also used during training.

3.2.1.3 Inference

Like training, only a single network feed-forward pass is required for predictions during test time. Since the network predicts many bounding boxes (approximately 98 boxes per image), many boxes may not detect any object or several bounding boxes

may detect the same object. Therefore, filtering is necessary to keep only the relevant boxes. Filtering is achieved in two steps: firstly, the boxes with confidence lower than a certain fixed threshold are removed and secondly an algorithm named *Non Max Suppression (NMS)* is used to deal with overlapping boxes.

Algorithm 12: NMS algorithm

Input: a set of proposal boxes $\mathbf{B}$, corresponding confidence scores $\mathbf{S}$ and overlap threshold t

Output: a set of filtered boxes $\mathbf{D}$

1 **while** $\mathbf{B}$ *is not empty* **do**

2 Select the box with highest confidence score, eliminate it from B and append it to the final box list $\mathbf{D}$ (Initially $\mathbf{D}$ is an empty list);

3 Compute the IOU of the selected box with every other box $\mathbf{b}$ in $\mathbf{B}$. If the IOU is greater than the threshold t, delete $\mathbf{b}$ from $\mathbf{B}$;

4 Pick again the box with the highest confidence from the remaining boxes in $\mathbf{B}$, eliminate it from $\mathbf{B}$ and add it to $\mathbf{D}$;

5 Once again compute the IOU of the new selected box with all the remaining boxes in $\mathbf{B}$ and remove the boxes which have high IOU than threshold t;

6 **return** $\mathbf{D}$

3.2.1.4 Limitations and further improvements

YOLO has difficulty to deal with nearby objects and especially small objects in groups and this is caused by strong spatial constraints imposed on bounding box predictions[33] (YOLO predicts only two bounding boxes per grid cell) . Moreover YOLO, struggles to generalize to detect objects in new or unusual aspect ratios or configurations (Over-fitting on training boxes).

Intending to resolve these problems, the author of YOLO proposed two improved versions of YOLO:

- **YOLOv2 or YOLO9000** [40]: YOLO underwent few transformations to become YOLOv2. These changes allow to improve training and increase performance. YOLOv2 introduced two main improvements. Firstly, the backbone was updated from GoogleNet to Darknet-19 [40] : a fully convolutional network with 19 convolutional layers containing batch normalization and five max-pooling layers. Secondly, the model does not predict the bounding boxes directly, but instead predicts scales and translates of them from predefined boxes called *anchor boxes* (see section 3.1.4). These *anchor boxes* are characterized by their width $\mathrm{p_w}$ and height $\mathrm{p_h}$ and are generated from the training data with *k-means* algorithm. The

idea of using anchor boxes is to get good priors for the model. The width and of height of a box is predicted as offsets from the anchor boxes. Moreover,one predicts the center coordinates of the box relative to the location of filter application using a sigmoid function [40]. Therefore, if the network predicts $\mathrm{t_x}$, $\mathrm{t_y}$, $\mathrm{t_w}$, $\mathrm{t_h}$ as box coordinates, the final predictions are calculated with the following equations:

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

where $\mathrm{c_x}$ and $\mathrm{c_y}$ are the cell offset coordinates from the top left corner of the image (figure 3.13).

if one uses a grid cells of shape $S \times S$, B anchor boxes and the training set contains C classes, the network output a shape $S \times S \times (B \times 5 + C)$ matrix. Finally, YOLOv2 is trained with the same cost function and has the same inference steps as YOLO (uses non-max suppression algorithm).

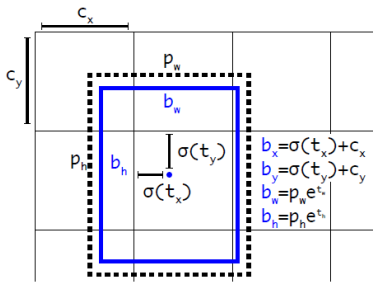


Figure 3.13: Bounding boxes with anchor boxes and location prediction [40]

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?								✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figure 3.14: The path from YOLO to YOLOv2 [40]

- **YOLOv3** [41] is an enhancement of [40] and is extremely faster and has higher accuracy. Like [40], YOLOv3 uses anchor boxes for bounding boxes prediction. The major difference with [40] is at the network output: with the aim of multi-scale object detection, YOLOv3 predicts boxes at 3 different scales (small, normal and high). Thus, if one uses a grid cells of shape $S \times S$ and the training dataset has C classes, the output shape is $S \times S \times [3 * (4 + 1 + C)]$. Moreover, YOLOv3 uses a deeper feature extractor (backbone). This backbone named darknet-53 (because it has 53 convolutional layers) [41] uses successive 3×3 and 1×1 convolutional layers and has some shortcut connections (Figure 3.15). Finally, YOLOv3 utilizes the same loss function used by [40] and inference steps remain the same.

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

Figure 3.15: Darknet-53 [41]

3.3 Two stage detectors

The task of detecting objects in images can be divided into two taks: i) proposal of regions and ii) classification + regression of the proposed regions. In the step of regions proposal, an algorithm is used to compute from the input image a set of regions (typically regions are boxes) which are likely to contain an object. The computed regions are called *Regions of Interest (ROI)*. Several image processing algorithms can be used to compute ROI. Howerver the most popular and efficient (in terms of best recall and speed) is *selective search* [42, 43]. Sometimes, instead of using selective search algorithm, a neural network named regions proposal network (RPN) is used to compute ROI.

Typically, two-stage detectors are regions based convolutional neural networks: R-CNN [44], fast-R-CNN [45], faster R-CNN [46] and mask R-CNN [47].

3.3.1 R-CNN

3.3.1.1 Principle

The R-CNN [44] model works in 4 steps:

1. Region proposals generation: selective search algorithm is used to generate approximately 2000 region proposals or regions of interest (ROI) from the input image.

2. Feature extraction: each region proposal is warped into a fixed resolution or shape and a pre-trained CNN on imagenet is fine-tuned to extract features from each region proposal. The last layer of the pre-trained CNN (the imagenet 1000-way classification layer) is replaced with randomly initialized (N+1)-way classification layer (N=number of classes +1 background) and then the obtained network is trained with labelled regions of proposal using stochastic gradient descent (SGD). Regions proposal are labelled with the following rule: all regions proposals with ≥ 0.5 IOU with a ground truth box are considered as positive for that box's class and the rest as negatives or backgrounds. During training, in each SGD iteration, a mini-bacth of size 128 with 32 positive region proposals (over all classes) and 96 negative windows is used to compute the loss. The bias sampling towards positive region proposals is to avoid over-fitting since positive region proposals are extremely rare compared to the negative ones.

In this step, we obtained the computed features from region proposals and their corresponding labelled classes.

3. Classification: The features obtained from region proposals and their corresponding labelled class are combined as an examples of a training set to train multiple support vector machines (SVM) for object classification (a SVM per class).

For training SMVs, only the ground-truth boxes are taken as positives example for their respective classes and labelled region proposals with len than 0.3 IOU overlap with all instances of a class as negative for that class. Proposals that have more than 0.3 IOU overlap but are not ground truth are ignored.

4. Bounding box regression: Linear regressors are used to perform bounding box regression form each labelled feature (One regressor per class).

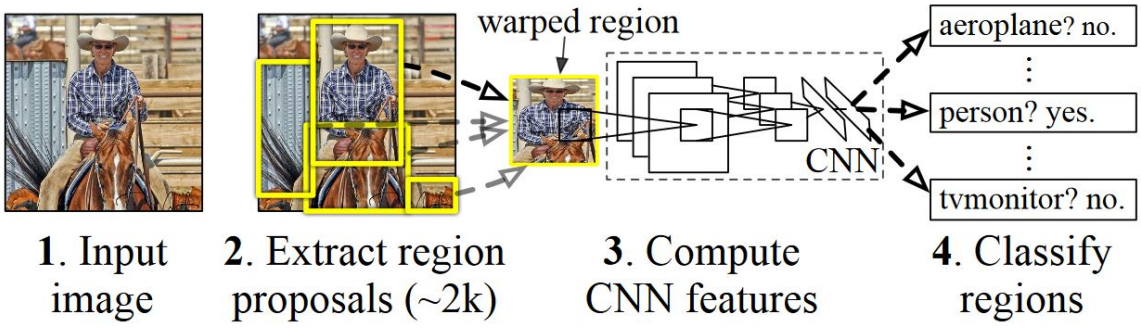


Figure 3.16: Overview of RCNN functioning [44]

3.3.1.2 Training

RCNN model is trained is 3 steps:
step 1: Domain-specific fine-tuning: Pre-trained network with softmax classifier (log loss) is fine-tuned.
step 2: Linear SVMs (hinge loss) are trained.
step 3: Bounding box regressors (squared loss) are trained.

3.3.1.3 Inference

During test time, given an input image, the following steps are followed to obtain predictions:
1. Selective Search is performed on the image to extract $2k$ regions proposals.

2. Warping each region proposal and computing feature vectors vector through CNN.
3. For each class, we score each extracted feature vector using the SVM trained for that class.
4. Given all scored regions in the image, for each class independently, we apply NMS algorithm which rejects a region if it has an IOU higher scoring selected region larger than a learned threshold.

R-CNN is more precise (higher mAP) than YOLO and SSD. However, because of its 3-step training, training is much more slow. Moreover, training R-CNN is memory space consuming and this is due to the need of storage in the disk features extracted from region proposals. Finally, due to selective search, inference is slow (47s/image).

3.3.2 Fast R-CNN

3.3.2.1 Basic functioning

R-CNN is time and space consuming because the model independently computes features for each region proposal. In fact, since region proposals have a high degree of overlap, independent features computation from each region proposal results in a huge volume of repetitive computations.

Fast R-CNN [45] resolves the problem of repetitive computations by computing a unique feature map from the entire image. Fast R-CNN takes as input the whole image and a set of region proposals (ROI). The network first computes a conv feature map from the image through a deep convNet (several conv and max pooling layers). Then, for each region proposal, a fixed size feature vector is extracted from the feature map with a region of interest (ROI) pooling layer. RoI pooling layer is a layer that uses max pooling to convert the features inside any region proposal into a small fixed shape feature map. Finally, each feature vector is sent to a sequence of fully connected layers which produces 2 output layers: one that is responsible for class probabilities prediction (produces softmax probabilities for all K categories $+1$ for background) and one that is responsible for bounding box regression (outputs for real values for each of the K object classes).

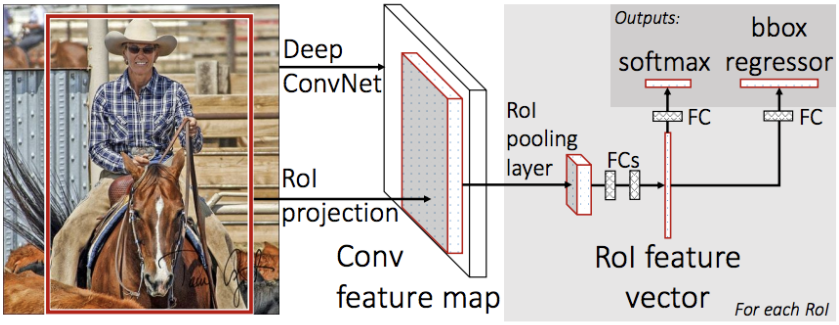


Figure 3.17: Overview of Fast R-CNN functioning [45]

3.3.2.2 Training

A pre-trained network on imagenet classification data set is turned into Fast R-CNN. Three transformations are necessary to turn a pre-trained netwrok for classification into Fast R-CNN : first, the last max pooling layer is changed to a RoI pooling layer; second the last layer of the network (1000-way ImageNet classification) is replaced by two sibling layers (one for class-probabilities prediction and one for bounding box regression); finally the network is modified to take two data inputs: a list of images and a list of region proposals in those images.

All the network weights are trained via a multi-task loss L in an end-to-end way. Like R-CNN region proposals annotation, each region proposal is labeled with a ground-truth class u and a ground-truth bounding-box regression target v. The multi-task loss L is used on each labeled region proposal to jointly train for classification and bounding-box regression:

$$L\left(p, u, t^u, v\right) = L_{\text{cls}}(p, u) + \lambda[u \geq 1] L_{\text{loc}}\left(t^u, v\right)$$

where $L_{\text{cls}}(p, u) = -\log p_u$ is the loss function for true class u and p_u is driven from the discrete probability distribution $p = (p_0, \ldots, p_K)$ over the $K+1$ outputs from the classification network. L_{loc} is the loss between true bounding-box coordinates $v = (v_{\text{x}}, v_{\text{y}}, v_{\text{w}}, v_{\text{h}})$ for a class u and predicted coordinates $t^u = \left(t^u_{\text{x}}, t^u_{\text{y}}, t^u_{\text{w}}, t^u_{\text{h}}\right)$ for u. The Iverson bracket indicator function $[u \geq 1]$ means there is no localisation loss for background ($u = 0$). A smooth L1 loss is used for L_{loc} :

$$L_{\text{loc}}\left(t^u, v\right) = \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}\left(t^u_i - v_i\right)$$

in which

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

3.3.2.3 Inference

To perform detection with a trained Fast R-CNN network, the network first take as input an image and a set of R region proposals (R is typically around 2000). For each region proposal r, the forward pass outputs a class posterior probability distribution p and a set of predicted bounding-box offsets relative to r (each of the K classes gets its own refined bounding-box prediction). Using the estimated probability

$$\Pr(\text{ class } = k \mid r) \triangleq p_k,$$

a detection confidence is assigned to r for each object class k. Then non-maximum suppression algorithm is perform independently for each class.

Fast R-CNN has higher detection quality (mAP) than R-CNN, YOLO and SSD but still uses selective search to find region proposals and remains slower.

3.3.3 Faster R-RCNN

3.3.3.1 Basic functioning and network architecture

Fast R-CNN requires selective search algorithm to generate many region proposals. With Faster R-CNN [46], selective search is replaced by a deep network called *region proposal network (RPN)*. This creates a faster and more accurate detection system.

The Faster R-CNN system is composed of two modules: the first module named RPN is a deep fully convolutional network that computes region proposals; and the second module is the Fast R-CNN network which uses the proposed regions.

RPN Network

The RPN takes an image of arbitrary size as input and outputs a set of rectangular object proposals, each with an objectness score:

• First, a deep CNN computes feature map from the input image.

• For each point in the feature map, the network has to predict the probability that an object is present in the input image at its corresponding location and also predict its size. This is achieved by putting a set of k anchor boxes of different sizes and aspect ratios on the input image corresponding to each pixel in the feature map. More precisely, for each pixel in the output feature map, the network has to verify if the

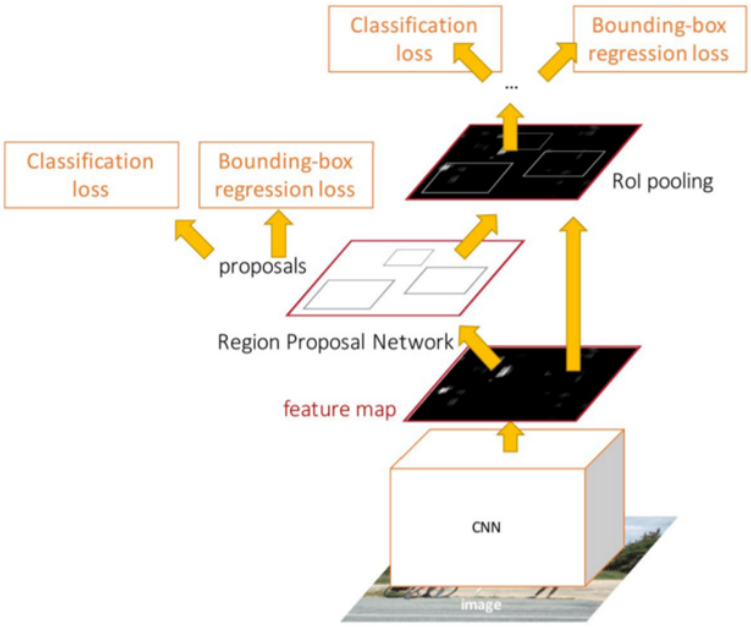


Figure 3.18: Faster R-CNN architecture [46]

placed anchor boxes contain object and then refine these anchors. A low dimensional vector (512-d) is computed for each pixel in feature map and sent to two sibling fully-connected layers: box-classification layer (cls) and box-regression layer (reg). This is achieved with an $n \times n$ conv layer followed by two sibling 1×1 conv layers.

3.3.3.2 Training

RPN Training

RPN is trained with labelled anchor boxes. Each anchor box is labelled with a binary class label of being object or not: anchor boxes with the higher IOU overlap with a ground-truth box or anchors with an IOU overlap higher than 0.7 with a groud-truth box are considered as positive (contain an object); anchors with IOU lower than 0.3 with all ground-truth boxes are negative (background); the other anchors are not used to train RPN. During training, the following loss function is optimized:

$$L\left(\{p_i\},\{t_i\}\right) = \frac{1}{N_{cls}} \sum_i L_{cls}\left(p_i, p_i^*\right)$$
$$+\lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}\left(t_i, t_i^*\right)$$

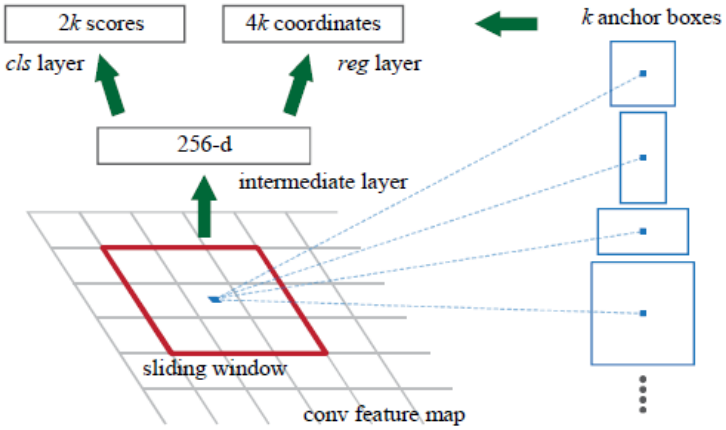


Figure 3.19: RPN: k predefined anchor boxes are use to predict region proposals and corresponding objectness score at each location. [46]

where:
• i is the index of an anchor box in a mini-batch and p_i the predicted probability that the anchor i is an object. p_i^* is the ground-truth label (1 if positive anchor and 0 if negative anchor).
• t_i is a 4-components vector and corresponds to predicted coordinates for anchor box i; and t_i^* is the ground truth vector coordinates. t_i^* is parametrized as following:

$$t_{\mathrm{x}}^* = \left(\mathrm{x}^* - \mathrm{x}_a\right)/w_a, \quad t_{\mathrm{y}}^* = \left(\mathrm{y}^* - \mathrm{y}_a\right)/h_a$$
$$t_w^* = \log\left(w^*/w_a\right), \quad t_h^* = \log\left(h^*/h_a\right)$$

where $\mathrm{x}*$, $\mathrm{y}*$, $w*$, $h*$ denote the center coordinates and the height and width of the ground-truth box corresponding to the anchor box with coordinates x_a, y_a, w_a and h_a. During test time, the x, y, w, h predicted coordinates for a bounding box can be back-calculated through the following parametrizations:

$$t_{\mathrm{x}} = \left(\mathrm{x} - \mathrm{x}_{\mathrm{a}}\right)/w_{\mathrm{a}}, \quad t_{\mathrm{y}} = \left(\mathrm{y} - \mathrm{y}_{\mathrm{a}}\right)/h_{\mathrm{a}}$$
$$t_{\mathrm{w}} = \log\left(w/w_{\mathrm{a}}\right), \quad t_{\mathrm{h}} = \log\left(h/h_{\mathrm{a}}\right)$$

- L_{cls} is a classification log loss over two classes (object or not). L_{reg} is a regression loss and a smooth $L1$ loss is used for it.

- N_{cls} and N_{reg} are normalization parameters. N_{cls} is equal to the mini-batch size and N_{reg} is equal to the number of anchor locations. λ is a weighted parameter (usually $\lambda = 10$).

RPN can be trained end-to-end with back-propagation and SGD. Each mini-batches arises from the same image and contains the same amount of positive and negative anchor boxes.

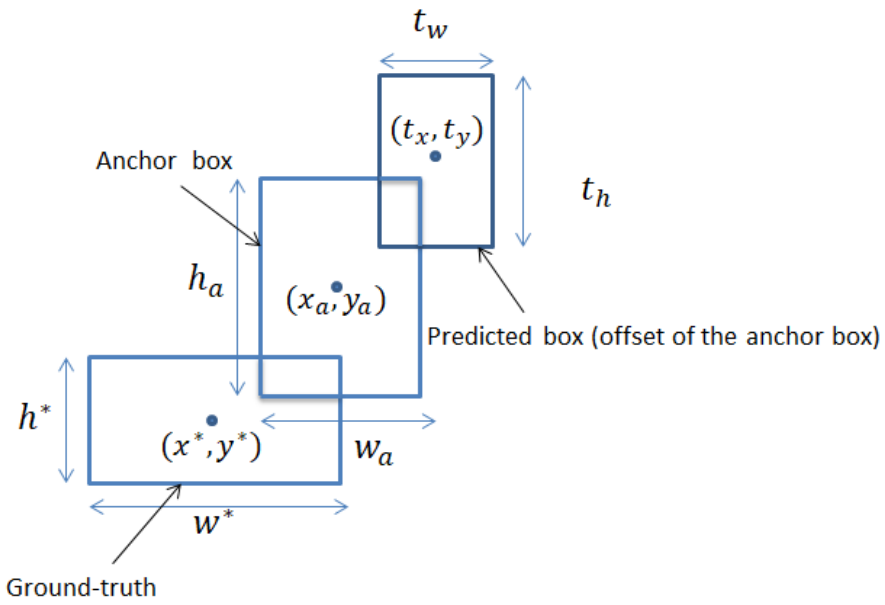


Figure 3.20: An anchor box, a predicted bounding box, and a ground truth box.

Faster R-CNN Training

The whole detection system is trained with a 4-step alternative training:
1.The RPN is trained independently.
2.The Fast R-CNN network is trained independently using the propsals generated by RPN.
3.The RPN is now initialized with weights from this Fast R-CNN, and fine-tuned for the region proposal task. This time, weights in the common layers between the RPN and detector remain fixed, and only the layers unique to the RPN are fine-tuned. This is the final RPN.
4.Finally, keeping the shared convolutional layers fixed, the unique layers of Fast R-CNN are fine-tuned.

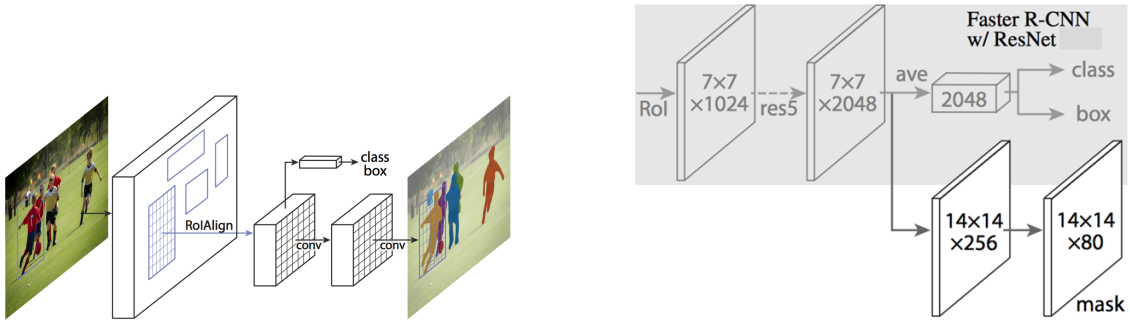
3.3.3.3 Inference

During test time, RPN predicts proposal boxes. To reduce redundancy (many proposals overlap), non-max suppression (NMS) is applied on the proposal regions based on their classification score. Usually, the IOU threshold for NMS is set to 0.7 which leads to 2000 region proposals per image. After NMS, top-N ranked proposals are used for detection.

3.3.4 Mask R-CNN

3.3.4.1 Basic functioning

Mask R-CNN [47] is an extension of Faster R-CNN for instance segmentation [49]. Faster R-CNN has two outputs for each region proposal: one for class prediction and another one for bounding box regression; Mask R-CNN adds a third output which predicts the object mask for instance segmentation. Like Faster R-CNN, Mask R-CNN is composed of two modules: the first module is RPN which proposes region proposals or region of interest (RoI); the second module proposes in addition to class prediction and bounding box regression, a binary output mask for each region proposal. For each RoI, the third module uses fully convolutional network (FCN) to predict K binary masks of resolution $m \times m$ which means that one mask for each of the K classes.



(a) Mask R-CNN architecture

(b) Head architecture

Figure 3.21: Mask R-CNN framework

3.3.4.2 Training

Mask R-CNN is trained with labelled regions of interest. RoI are labelled as in Fast R-CNN: an RoI is considered as a positive sample if it has IoU with a ground-truth box greater or equal to 0.5 and negative sample otherwise. During training, the following multi-tasks loss is minimized:

$$L = L_{cls} + L_{box} + L_{mask}$$

where the classification loss L_{cls} and the bounding box regression loss L_{box} are identical as those defined previously is Fast R-CNN. L_{mask} is defined as binary cross-entropy loss. The binary cross-entropy function computes the loss on a batch of size n by computing the following average:

$$\mathcal{L}(\mathrm{y}_i, \hat{\mathrm{y}}_i) = -\frac{1}{n}\sum_{i=1}^{n}\{\mathrm{y}_i \cdot \log \hat{\mathrm{y}}_i + (1-\mathrm{y}_i) \cdot \log (1-\hat{\mathrm{y}}_i)\}$$

where y_i is a binary target (0 or 1) and $\hat{\mathrm{y}}_i$ is class probability predicted by the network. Given an RoI labelled with a ground-truth class k, L_{mask} is only defined on the k^{th} mask which means that the other masks do not contribute to the loss.

3.3.4.3 Inference

At test time, RPN proposes a certain number of region proposals. Box prediction is performed with these proposals followed by non-max suppression. The mask branch then computes masks on the top 100 detection boxes. Finally, the floating-number mask output of resolution $m \times m$ is resized to the RoI size, and binarized at a threshold of 0.5.

Chapter 4

EVALUATION OF THE STUDIED METHODS ON OUR DATABASES

In this chapter, we describe and discuss the results we obtained by applying the object detection algorithms we presented so far on our datasets.

We firstly present our datasets and how we processed them for object detection algorithms. Then we present the results obtained by benchmarking object detection methods on those datasets. We evaluated five detection models with our datasets: YOLO, RetinaNet, Faster R-CNN, mask-RCNN and Cascade R-CNN [48].

4.1 Preparing data for object detection algorithms

4.1.1 Datasets

Here we present our two datasets with which we trained and evaluated detection algorithms. We used two datasets: *Rocknet* and *Foraminifera*.

- *Rocknet* is a dataset of RGB images containing rocks (Figure 4.6). Our objects of interest are rocks present in those images. Our goal is to detect and precisely localize all rock instances in the images.

- *Foraminifera* is a dataset containing scanned images of rock samples (Table 4.2). Those images contain micro-fossils which are our objects of interest. Our intention is to detect and recognize (class category) all objects of interest. The dataset contains nines types of micro-fossils which means that there exits 9 class categories in the dataset.

Table 4.1: Some images from Rocknet

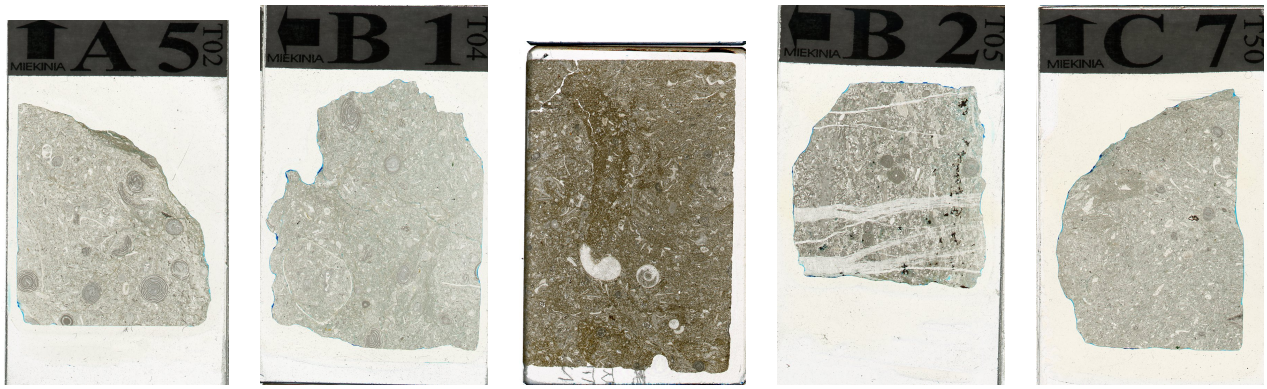


Table 4.2: Some images from Foraminifera

4.1.2 Bounding box annotation

Because detection systems are supervised learning algorithms, images must be labelled. Since, the goal is to find, localize and recognize objects of interest present in images, annotation is done by firstly providing coordinates of bounding boxes recovering the objects of interest and secondly setting the class category of each object of interest (Figure 4.3). To annotate our images, we utilized an annotation tool named `Microsoft VoTT` [1] (Figures 4.1 and 4.2).

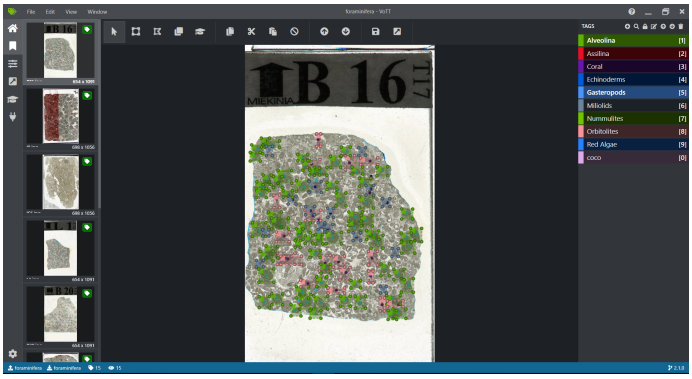


Figure 4.1: Foraminifera annotation with VoTT: We putted rectangular boxes on each object of interest (here micro-fossil) and assigned each box with a one of the nine class labels (See Top right of the figure).

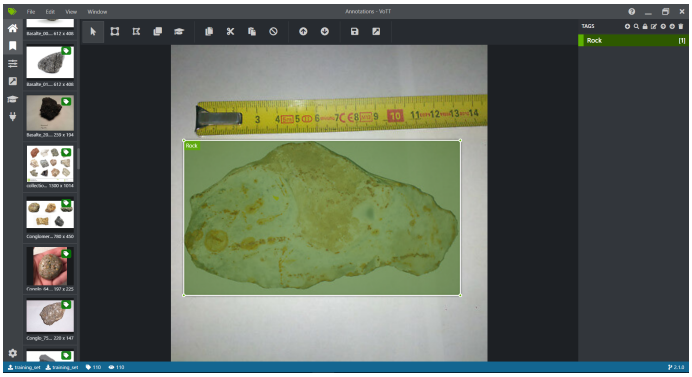


Figure 4.2: Rocknet annotation with VoTT: We have only one class label in the dataset.

[1] https://github.com/microsoft/VoTT

(a) Labelled image from Foraminifera

(b) Labelled image from Rocknet

Figure 4.3: Annotated images with VoTT

4.1.3 Anchor boxes generation

Since YOLO, Faster R-CNN, Mask R-CNN and Cascade R-CNN use anchor boxes, we utilized *k-means* to generate five anchor boxes from each of the two datasets. These anchor boxes will serve as prior knowledge for bounding box prediction. The following two images (Figues 4.5 and 4.4) show the five prior boxes we generated from Rocknet and Foraminifera respectively.

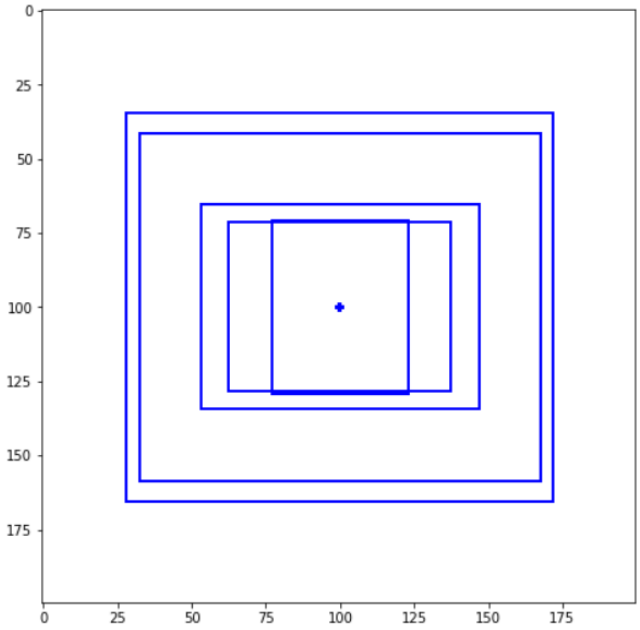


Figure 4.4: Anchor boxes from Rocknet

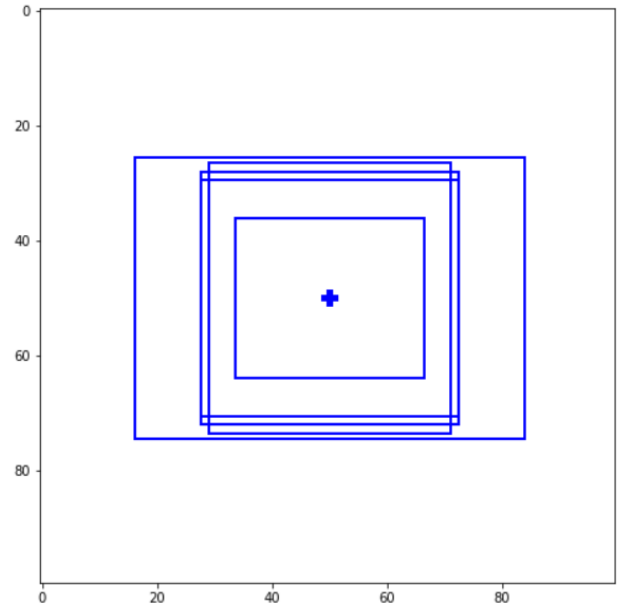


Figure 4.5: Anchor boxes from Foraminifera

4.2 Results

We trained the detectors using *Transfer Learning*. Transfer Learning is a machine learning technique where knowledge gained while solving one problem is exploited to solve different problems. Transfer Learning is very crucial when we deal with problems with limited training data. [50, 51] proved the efficiency of Transfer Learning for solving small dataset problems. Transfer Learning consist of two steps: **Pre-Training** and **Fine Tuning**.

Pre-Training
The network is trained on a large dataset (benchmark datasets: ImageNet, COCO dataset [52], etc.) and all the parameters of the model are saved.

Fine-tuning
We firstly remove the final layers of the pre-trained network and replace them by new layers adapted to our task. Then we can either retrain only the new layers while keeping the others fixed or retrain the whole model with initial weights from the pre-trained network.

For our detection task, the networks are pre-trained on COCO dataset. Let's now present the results we obtained for our two datasets.

4.2.1 Rocknet

We trained YOLO[2], RetinaNet[3], Faster R-CNN[3], Mask R-CNN[3] and Cascade R-CNN[3] on a training set of 700 images and evaluated them on a test set of 290 images. To evaluate the detectors, we used the mAP metric from [23] repository. The table below presents the results we obtained.

Evaluation of detectors on RockNet			
Detector	Precision (AP50)	Precision (AP75)	inference speed on CPU
YOLOv3	96.79	80.89	0.77 s / img
RetinaNet	96.64	86.49	3.2 s / img
Faster R-CNN	98.54	89.25	4.19 s / img
Mask R-CNN	98.43	86.49	5.96 s / img
Cascade R-CNN	98.61	86.30	5.62 s / img

[2]https://github.com/AntonMu/TrainYourOwnYOLO
[3]https://github.com/facebookresearch/detectron2

In the above Table, AP50 and AP75 mean that the threshold is set to 0.5 and 0.75 respectively. We observed that one-step methods (YOLOv3 and RetinaNet) are somewhat less accurate but faster than multi-step methods (Faster R-CNN, Mask R-CNN and Cascade R-CNN). We also noticed that Faster R-CNN is slightly more accurate than Mask R-CNN despite the similarity of their architectures. This is explained by the fact that Mask R-CNN has a segmentation additional branch which degrades object detection performance. Definitely, the results obtained for this dataset are very satisfactory.

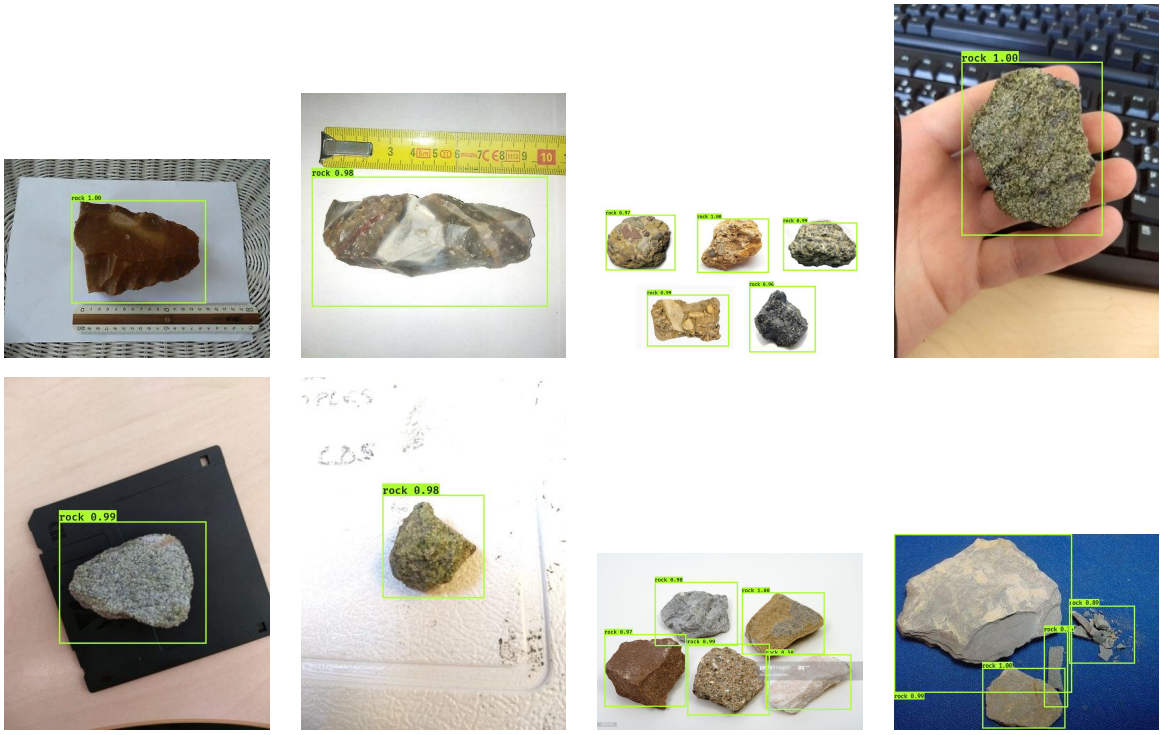


Figure 4.6: Some detections from Rocknet.

4.2.2 Foraminifera

With Foraminifera, we encountered the problem of limited data and imbalanced classes (Figure 4.7). In fact, we had only 15 annotated images. Using data augmentation, we trained the five networks with the 15 images and evaluated them with the same training 15 images. Due to a lack of labelled data, evaluation on test data is

done qualitatively by geological experts and apart from YOLO, the results obtained with the other detectors are very promising. Evaluation results on the training set are presented in the Figure 4.8. These results are very good however since they are obtained on the training set, there is a risk of overfitting. More annotated data is needed for training and validation.

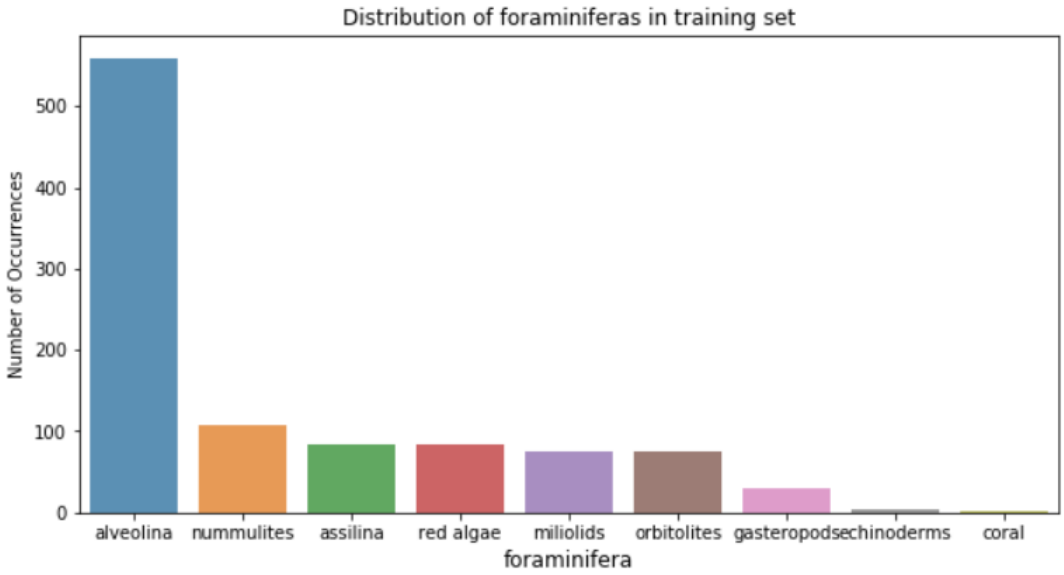
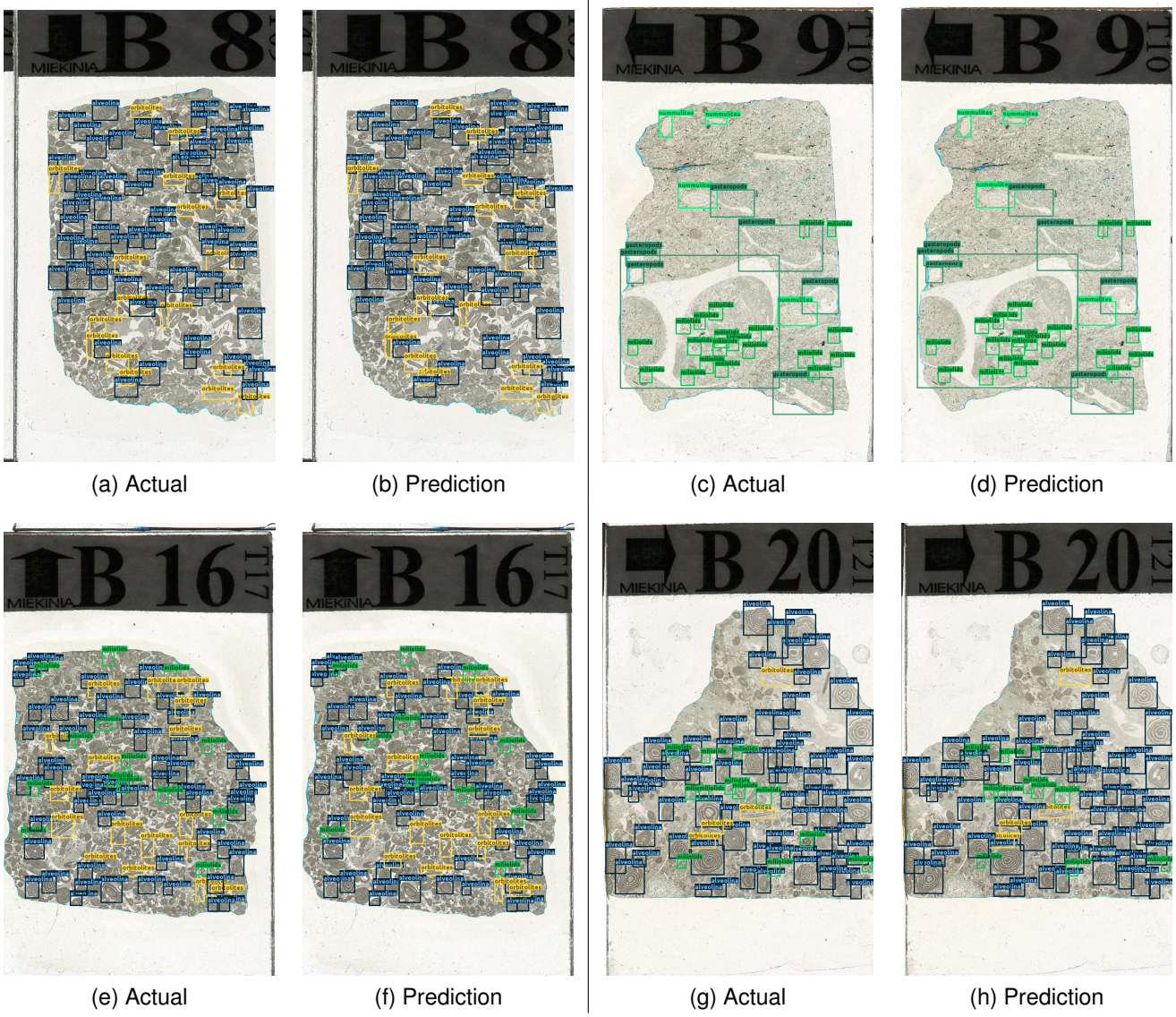


Figure 4.7: Distribution of foraminifera classes in the training data

Evaluation of detectors on Foraminifera			
Detector	Precision (AP50)	Precision (AP75)	inference speed on CPU
YOLOv3	/	/	/
RetinaNet	88.095	87.73	7.10 s / img
Faster R-CNN	94.087	93.07	11 s / img
Mask R-CNN	96.36	86.74	12.43 s / img
Cascade R-CNN	95.60	94.96	12.45 s / img

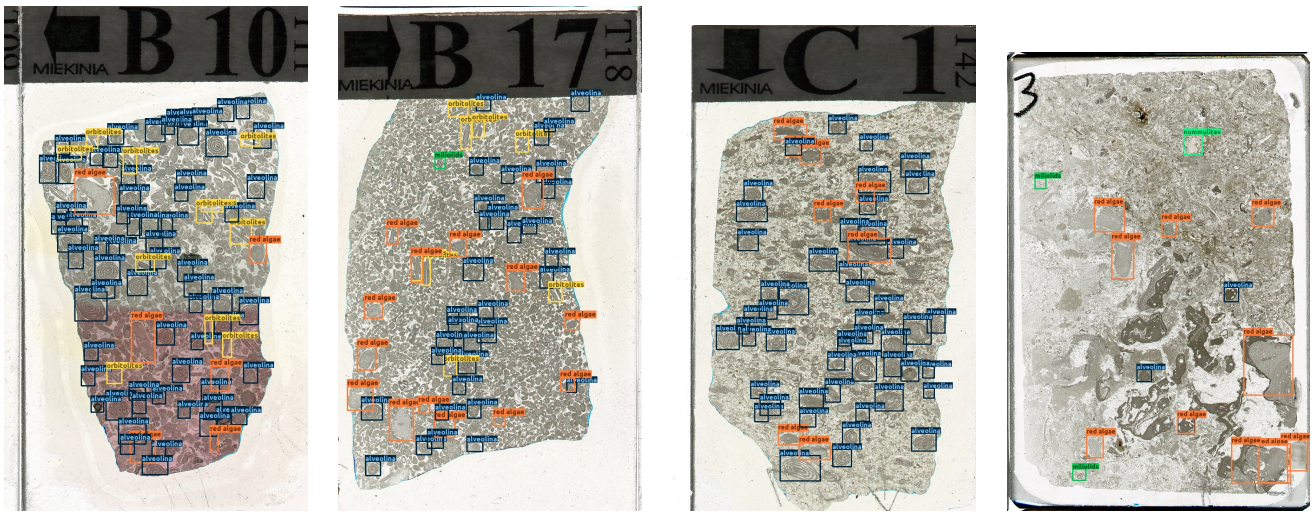
Figure 4.8: Results obtained for Foraminifera. Evaluation is done with training data



(a) Actual (b) Prediction (c) Actual (d) Prediction

(e) Actual (f) Prediction (g) Actual (h) Prediction

Table 4.3: Foraminifera: Some detections from training images with Mask R-CNN.



(a) (b) (c) (d)

(e) (f) (g) (h)

Table 4.4: Foraminifera: Some detections from test images with Mask R-CNN.

Chapter 5

CONCLUSION AND PERSPECTIVES

5.1 Conclusion

We used computer vision and deep learning approaches for geological applications. Specifically we utilized deep learning based object detection techniques for rock detection and automatic identification of micro-fossils.

Deep learning object detection methods can be classified into two groups: one-step methods (YOLO, SSD, RetinaNet) and multi-step methods (R-CNN, Fast R-CNN, Faster R-CNN, Mask R-CNN, Cascade R-CNN). The one-step methods are faster but less accurate than the multi-step methods.

We evaluated with our databases YOLO, RetinaNet, Faster R-CNN, Mask R-CNN and Cascade R-CNN. The results obtained for rock detection are very satisfactory. For the automatic identification of micro-fossils, the results are less satisfactory due to the lack of sufficient training data.

5.2 Perspectives

Rock detection by deep learning and computer vision will be integrated into the internet RockNet project for robust rock samples characterization.

The results obtained for the automatic identification of micro-fossils constitute for us a prototype that proves the feasibility of automatic detection of very small micro-fossils with deep learning and computer vision approaches. Future work will be carried out to improve the results obtained. In particular, object detectors will be trained with many more images acquired and annotated by geological experts. Moreover, we are not interested in the location and size of micro-fossils in images, but rather only in the

distribution of micro-fossil types. Therefore in future work, the architecture of existing detection methods may be modified to identify micro-fossils by points (Figure 5.1). This could help to have a faster and more robust system.

Finally, future work will focus on predicting specific localization forms (circles, ellipses) instead of bounding boxes (Figure 5.2). These types of detections have very important geological applications, in particular the detection of the shapes of quartz grains for evaluation of geological deformations of sedimentary environments.

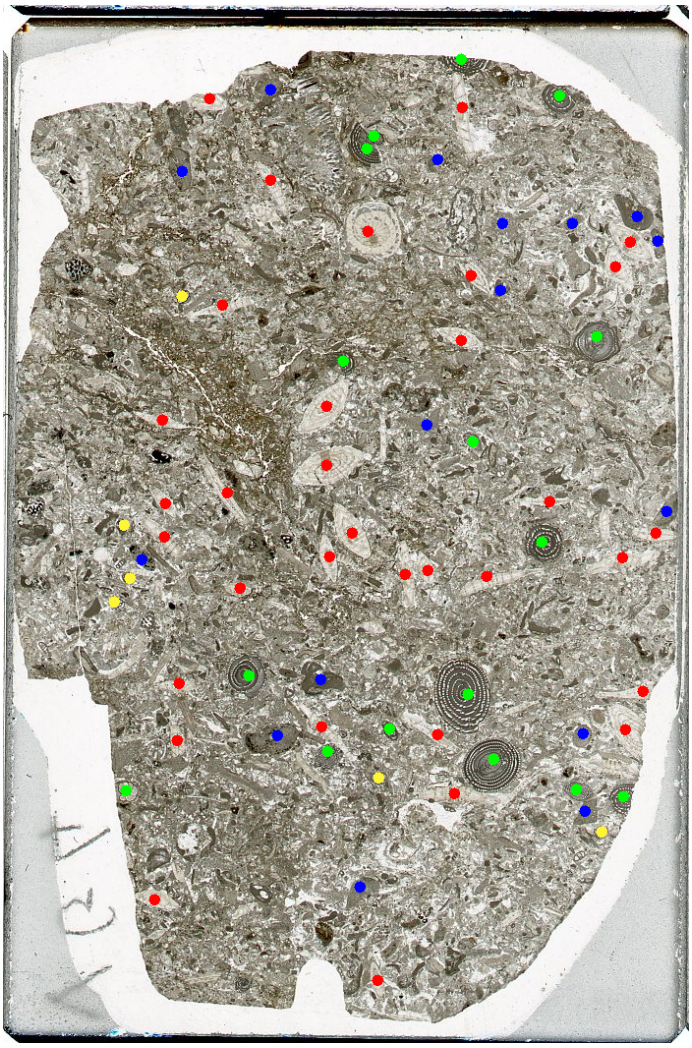
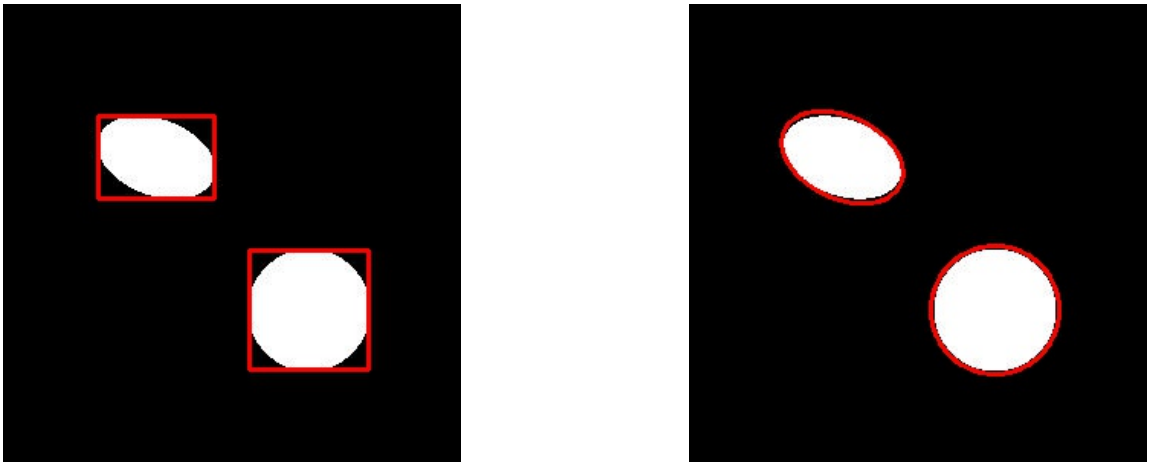


Figure 5.1: Micro-fossils identified by points



(a) Current location (b) Our perspective

Figure 5.2: Illustration of our approach

Chapter 6

Bibliography

[1] A. Bhandare, M. Bhide, P. Gokhale, R. Chandavarkar, *Applications of Convolutional Neural Networks*, IJCSIT, 2016, vol 7, n.5

[2] Z. Zhao, P. Zheng,S. Xu, X. Wu, *Object Detection with Deep Learning: A Review*, `arXiv:1807.05511v2`

[3] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*

[4] T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning, Springer*

[5] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola,*Dive into Deep Learning,Release 0.7.0*

[6] B. Krose, P. van der Smagt, *An introduction to Neural Networks*

[7] N. M. Mishachev, *BACKPROPAGATION IN MATRIX NOTATION*

[8] Andrew Ng's deep learning specialization course on coursera

[9] Y. LeCan, L. Bottou, Y. Bengio and P. Haffner, *Gradient-Based Learning applied to Document Recognition*

[10] A. Krizhevsky, I. Sutskever, G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*

[11] K. Simonyan and A. Zisserman, *VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*

[12] K. He, X. Zhang, S. Ren and J. Sun *Deep Residual Learning for Image Recognition*

[13] S. Ioffe and C. Szegedy , *Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.*

[14] J. Zhang, *Gradient Descent based Optimization Algorithms for Deep Learning Models Training*

[15] S. Ruder ,*An overview of gradient descent optimization algorithms*

[16] N. Qian, *On the momentum term in gradient descent learning algorithms. Neural Netw., 12(1):145151, January 1999.*

[17] Y. Nesterov, *A method of solving a convex programming problem with convergence rate O(1/sqr(k)). Soviet Mathematics Doklady, 27:372–376, 1983.*

[18] J. Duchi, E. Hazan, and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res., 12:2121–2159, July 2011.*

[19] T. Tieleman and G. Hinton, *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.*

[20] *Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. CoRR, abs/1212.5701, 2012.*

[21] Diederik P. Kingma and J. Ba, *Adam: A method for stochastic optimization. CoRR, abs/1412.6980, 2014*

[22] S. Agarwal, J. O. du Terrail, F. Jurie, *Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks*

[23] R. Padilla, S. L. Netto and E. A. B. da Silva, *Survey on Performance Metrics for Object-Detection Algorithms, International Conference on Systems, Signals and Image Processing (IWSSIP) 2020*

[24] J. Davis, M. Goadrich, *The Relationship Between Precision-Recall and ROC Curves*, http://pages.cs.wisc.edu/ jdavis/davisgoadrichcamera2.pdf

[25] The PASCAL Visual Object Classes (VOC) Challenge

[26] B. Ding, H. Qian and J. Zhou, *Activation functions and their characteristics in deep neural networks, 2018 Chinese Control And Decision Conference (CCDC), Shenyang, 2018, pp. 1836-1841, doi: 10.1109/CCDC.2018.8407425.*

[27] He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. arXiv:1502.01852

[28] H. Geoffrey, D. Li, Y. Dong, D. George, M. Abdel-rahman, J. Navdeep, S. Andrew, V. Vincent, N. Patrick, S. Tara, K. Brian (2012). *Deep Neural Networks for Acoustic Modeling in Speech Recognition.*

[29] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio Deep sparse rectifier neural networks International Conference on Artificial Intelligence and Statistics (2011)

[30] Y. LeCun, Y. Bengio, *Convolutional Networks for Images, Speech, and Time-Series*

[31] D. Scherer, A. Muller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in Proc. of the Intl. Conf. on Artificial Neural Networks, 2010, pp. 92–101.

[32] P. Hurtik, V. Molek, J. Hula, M. Vajgl, P. Vlasanek and T. Nejezchleba, *POLY-YOLO: HIGHER SPEED, MORE PRECISE DETECTION AND INSTANCE SEGMENTATION FOR YOLOV3*

[33] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, `arXiv:1506.02640`

[34] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C; Fu, and A. C Berg, *Ssd: Single shot multibox detector. In European conference on computer vision, pages 21–37. Springer, 2016.*

[35] T. Lin, P. Goyal, R. Girshick, K. He, P. Dollar, *Focal Loss for Dense Object Detection*

[36] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, *V. Vanhoucke, and A. Rabinovich, Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition*

[37] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 2015*

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*

[39] N. Bodla, B. Singh, R. Chellappa, L. S. Davis, *Improving Object Detection With One Line of Code*

[40] J. Redmon, A. Farhadi, *YOLO9000: Better, Faster, Stronger*

[41] J. Redmon, A. Farhadi, *YOLOv3: An Incremental Improvement*

[42] Jasper R. Uijlings, K. E. Van De Sande, T. Gevers, and A. WM Smeulders, *Selective search for object recognition. International Journal of Computer Vi- sion (IJCV), 104(2):154171, 2013.*

[43] K. E. Van de Sande, J. R. Uijlings, T. Gevers, and A. WM Smeulders, *Segmentation as selective search for object recognition. In IEEE International Conference on Computer Vision, ICCV 2011, Barcelona, Spain, November 6-13, 2011.*

[44] R. Girshick, J. Donahue, T. Darrell, J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, `arXiv:1311.2524`

[45] R. Girshick, *Fast R-CNN*, `arXiv:1504.08083`

[46] S. Ren, K. He, R. Girshick, J. Sun *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, `arXiv:1506.01497`

[47] K. He, G. Gkioxari, P. Dollar and R. Girshick, *Mask R-CNN*

[48] Z. Cai, N. Vasconcelos, *Cascade R-CNN: Delving into High Quality Object Detection*

[49] A. Arnab and P. H.S Torr, *Pixelwise Instance Segmentation with a Dynamically Instantiated Network*

[50] Z. Kolar, H. Chen, and X. Luo, *Transfer learning and deep convolutional neural networks for safety guardrail detection in 2D images.*

[51] Y. Gao and K. M. Mosalam, *Deep Transfer Learning for Image Based Structural Damage Recognition.*

[52] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollr, and C. L. Zitnick, *Microsoft COCO: Common objects in context*.