UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

Master's Degree in Automation Engineering

3D FEATURE EXTRACTION AND OBJECT DETECTION IN POINT CLOUDS

Student	*Supervisor*	*Co-supervisors*
Alberto Adami	**Prof. Ruggero Carli**	**Roberto Polesel**
		Walter Zanette
		(from Euclid Labs)

JULY 8, 2019

ACADEMIC YEAR 2018/2019

Contents

Abstract

The final goal of this work consists of realizing a model able to recognize groups of objects, sparsely distributed inside a container. Such model must take as input a 3-dimensional array of x-y-z coordinates, namely a point cloud that faithfully represent the targets in the 3D space, acquired through a Scanner.
In order to achieve the goal, a modified version of Yolo is proposed; it consists of a fast algorithm for detection in RGB images, here adapted to deal with 3D point clouds. The main difficulty is the identification of some features able to univocally describe each part of the cloud. Some features could be local point density, variance, curvature, linearity, space distibution and others. The first part of the work is going to describe some methods to extract such features; the last part will describe Yolo model and how to use such features to make Yolo working for our target.
Yolo model consists of a sequence of Convolutional Neural Networks (CNN). It takes as input the preprocessed point cloud, subdivided in voxels intervals along x-y axis, each one including all the cloud points with x-y coordinates falling in each interval. Then, for each voxel, the maximum height is extracted in such a way to obtain the point cloud z-image, which will feed the CNNs. Each voxels represents some local features that identifies a specific area of the object.
For the training, many randomly rotated and translated version of the target object were generated. It was also added some background in order to faithfully replicate the environment and make the model able to discriminate background from foreground.

0.1 Euclid Labs

[1] This work is done in collaboration with Euclid Labs, a company situated in Nervesa della Battaglia (Tv, Italy).
Such company designs and develops hi-tech solutions for robotics and industrial automation. They deal with the common hi-tech problems in the fields of automation and machine learning, such like 3D vision, random bin picking and robot programming.

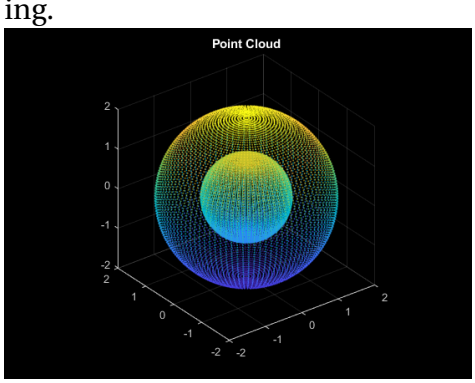
Chapter 1

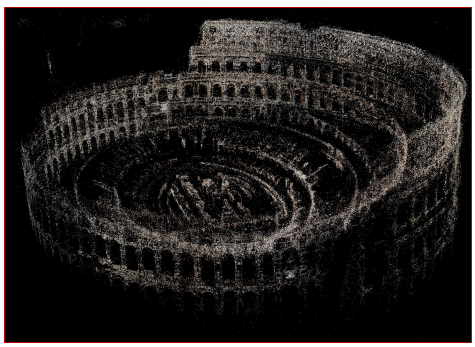
Introduction

1.1 Point clouds

Let's start from the definition of Point Cloud: a Point Cloud is a set of data points in space defined by a given coordinates system; each point is characterized by xyz coordinates and, eventually, from color intensity (e.g. RGB color spaces).
Point clouds are used in many fields of information engineering, such as computer vision, machine learning, deep learning and object recognition, and many field of technology like construction, quality evaluation and others.
These data present a valid alternative to images, with many advantages compared to them. One of these is the possibility to deal with 3D coordinates, adding useful information for the model we want to train. Moreover, point clouds offer the possibility to manage large amounts of data (order of million points).

A brief introduction to point clouds has been provided; now, how can we manage this interesting dataset in order to perform good object detection? Let's explain a brief theoretical background about machine learning and deep learning.





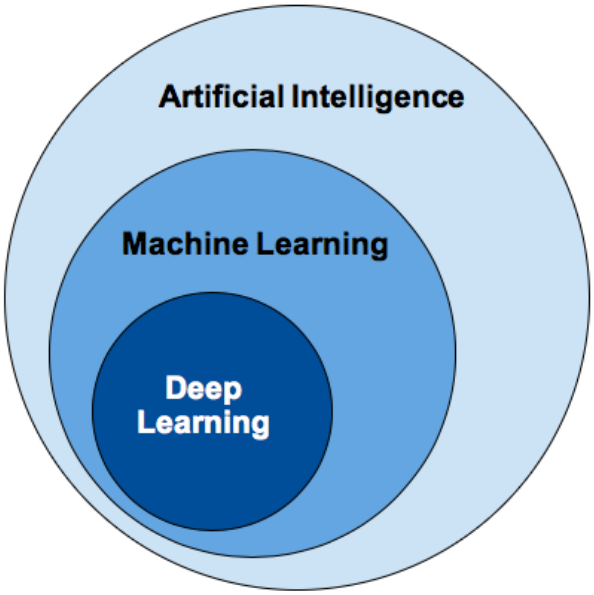
1.2 Machine learning and deep learning

Machine learning can be described starting from its basic definition:

"Algorithms that parse data, learn from that data, and then apply what they have learned to make informed decisions."

Basic machine learning models do become progressively better at whatever their function is, but they still need some guidance. If an ML algorithm returns an inaccurate prediction, then it's needed to step in and make adjustments from the outsides.

What about deep learning?
Deep learning is just a subset of machine learning. It technically is machine learning and functions in a similar way, but its capabilities are different. Indeed, with a deep learning model, the algorithms can determine on their own if a prediction is accurate or not.



Chapter 2

Machine learning techniques

2.1 Feature Extraction

2.1.1 2D Point clouds

Point sorting

Before selecting and extracting features, it may be useful to make a point sorting. The point cloud could be provided without a precise order of the points, which may be randomly shuffled.
The idea is to make a sort based on the point distances. The steps are the following:

1. consider the first point of the cloud, and save it on the first position of a parallel array;

2. find the point whit minimum distance from the point considered;

3. eliminate the first point from the original point cloud array and save the point found at step 2 in the second position of the parallel array;

4. now, the point cloud array contains $N-1$ points (N = total number of points in the cloud). Repeat from step 1 and update the parallel array from the first free position.

5. Stop when the original array is empty.

At the end, the parallel array will contain all the points sorted by distance.
It's easy to note that this is a recursive function, with N recursive calls.

Local Variance and Structure Tensor

In order to get more confidence with point clouds, some trials were initially performed on simple clouds, representing circles, squares and triangles; the goal was to classify them.
According to some papers, the local feature we decided to use is the local variance of each point:

1. For each point of the cloud, a k-neighborhood was computed, based on points distances.

2. For each point, the structure tensor was computed, namely the covariance matrix, who's eigenvalues and eigenvectors describe the three direction of the distribution of the points in its neighborhood.
   This 3x3 matrix is computed from the discrete gradients along x, y, z:

$$I_{ij_x} = p_{i_x} - r_{j_x}$$

$$I_{ij_y} = p_{i_y} - r_{j_y}$$

$$I_{ij_z} = p_{i_z} - r_{j_z}$$

   where p is a point of the cloud and r is a point of its neighborhood. The gradients are computed for each i ϵ [0, |point cloud|], j ϵ [0, |neighborhood of p|]. At this point, for each point in the cloud, there are k gradients, with k corresponding to the number of points in the neighborhood.
   In order to obtain the p_i Structure Tensor S_i, we need to compute the following:

$$S_{i_{xx}} = \Sigma_{j=0}^{k} I_{ij_x}^2$$

$$S_{i_{yy}} = \Sigma_{j=0}^{k} I_{ij_y}^2$$

$$S_{i_{zz}} = \Sigma_{j=0}^{k} I_{ij_z}^2$$

$$S_{i_{xy}} = \Sigma_{j=0}^{k} I_{ij_x} * I_{ij_y}$$

$$S_{i_{xz}} = \Sigma_{j=0}^{k} I_{ij_x} * I_{ij_z}$$

$$S_{i_{yz}} = \Sigma_{j=0}^{k} I_{ij_y} * I_{ij_z}$$

   The structure tensor is the following:

$$S_i = \begin{bmatrix} S_{i_{xx}} & S_{i_{xy}} & S_{i_{xz}} \\ S_{i_{xy}} & S_{i_{yy}} & S_{i_{yz}} \\ S_{i_{xz}} & S_{i_{yz}} & S_{i_{zz}} \end{bmatrix}$$

Feature selection

From matrix S_i, eigenvalues λ_1, λ_2, λ_3 and eigenvectors v_1, v_2, v_3 indicate the three main directions of the distribution of the points inside the neighborhood. For our purpose, a particular value, named M_i, was selected:

$$M_i = det(S_i) - k * trace(S_i)$$

It derives from the theory of Harris corner detection; it is able to efficiently discriminate the curvature of the neighborhood, giving information if we are in proximity of a corner, edge or neither. k is equal to 0.04.

2.1.2 3D point clouds

Once 2D point clouds are analyzed and got more confident with managing this type of data, we can step to next level: extract feature of 3D point clouds in order to classify them.
The clouds we consider are:

- Spheres
- Cylinders
- Planes

The initial idea for the feature extraction was the same procedure for the 2D, namely compute the local structure tensor of each point of the cloud. This method leaded to poor results. Indeed, the extracted parameter M_i along the 3D surface of the cloud was not distributed following a precise function, as in the 2D case. Hence, the idea was to project the 3D point cloud along the two main axis along the points are distributed. This operation takes the name of Principal Component Analysis and it is applied before the feature extraction.

Projection in 2D with Principal Component Analysis

Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

The three directions of the variance are given by the eigenvectors of the covariance matrix of the 3D point cloud array:

$$\Sigma = \frac{1}{N}\begin{bmatrix} (X-\mu_x)^T(X-\mu_x) & (X-\mu_x)^T(Y-\mu_y) & (X-\mu_x)^T(Z-\mu_z) \\ (Y-\mu_y)^T(X-\mu_x) & (Y-\mu_y)^T(Y-\mu_y) & (Y-\mu_y)^T(Z-\mu_z) \\ (Z-\mu_z)^T(X-\mu_x) & (Z-\mu_z)^T(Y-\mu_y) & (Z-\mu_z)^T(Z-\mu_z) \end{bmatrix}$$

where:
X = x-coordinates of cloud points
Y = y-coordinates of cloud points
Z = z-coordinates of cloud points
μ_x, μ_y, μ_z = mean value of X, Y, Z
N = point cloud cardinality

The eigenvector corresponding to the largest eigenvalue of the covariance matrix, named First Component, represents the main direction of the point cloud variance. The other components are named Second Component and Third Component.
The idea is to project the point cloud along first and second components, rejecting the third; in this way, it is possible to obtain a good projected 2D point cloud, with the lowest loss of information from the 3D one.
Using the principal components, the projection can be obtained in the following way:

1. Compute the x and y local projection of the point cloud, compared to the two main eigenvectors:

$$X_{localproj} = [X, Y, Z]\,v1$$

$$Y_{localproj} = [X, Y, Z]\,v2$$

2. Compute the local projected points:

$$P_{local} = \begin{bmatrix} X_{localproj} & Y_{localproj} & 0 \end{bmatrix}$$

3. Compute the final projection of the point cloud, compared to the canonical axis:

$$P_{final} = M_{proj} P_{local}$$

where:

$$M_{proj} = \begin{bmatrix} v1 & v2 & v3 \end{bmatrix}$$

Feature extraction and selection

Once we obtained the projected point clouds, it is possible to extract the features applying the same method exploiting the M parameter of the structure tensor S.

2.2 Classification

The classification problem is a supervised learning technique that allows to split some initially sparse data into clusters, according to some discriminating features.

2.2.1 Introduction to classification models

In machine learning, once the features are extracted, they feed a classification model, which could be of different types. The best, and the one we will use, is the fully connected neural network.
The fully connected neural network belongs to the set of the artificial neural networks (ANN). ANNs are inspired to the brain neural network, which consists, briefly, on a huge number of neurons interconnected between each other; the ANN exploit the same principle: the neurons are called 'nodes' and they are organized in layers, which are splitted into input, hidden and output layer.
Each node is modelized by an activation function, which will be explained in details in 'YOLO' chapter.
Let's see how to exploit such powerful model with our point clouds.

2.2.2 Point clouds classification

The M_i parameter, $i \epsilon [0, |pointcloud|]$, is used as input for the classification model. Such model consists of a simple fully connected neural network, composed by an input layer, one hidden layer and 3 outputs, each one representing the probability score the cloud we are considering is a circle, a triangle or a square.
The input layer is composed by a number nodes equal to the number of points of the cloud considered; each node stores an extracted feature M_i. Then, there are two fully connected hidden layers, which elaborate the features extracted and finally an output layer. The latter contains many nodes as the number of classes we want to detect; in this case, the number of classes is 3, so the output nodes number will be 3. Each one of these nodes, stores a value, which indicates the probability score of each class.

2.2.3 Loss function

The model loss function is a sparse categorical crossentropy loss:

$$L(w) = -\sum_{i=1}^{N} y_{true,i} \log(y_{pred,i})$$

where: w stores the model parameters, namely the weights;
y_{true} stores the true classification labels;
y_{pred} stores the model output predictions. The goal is to find the right w, such that the loss function is minimum, namely y_{pred} equals y_{true}.

2.2.4 Training and Testing

The training operation consists on gradually updating the weights parameters w in such a way to minimize the loss function. In practice, it consists on finding the best prediction y_{pred} such that $y_{pred} = y_{true}$. To achieve a result, the loss function must be, of course, differentiable and convex.
During the training, the model (namely, feature extractor + 'neural network' classifier) is fed by $15K$ point clouds, arranged as it follows:

- $5K$ spheres;
- $5K$ cylinders;
- $5K$ planes.

All point clouds are feature extracted once and then feed the classifier for 5 training epochs.
After the training, a testing process is performed, in order to verify that features were well-extracted and prevent an eventual case of overfitting. During the testing, new fresh data feed the model, with the task to verify if they are classified correctly. Such fresh data are organized as it follows:

- 1500 spheres;
- 1500 cylinders;
- 1500 planes.

If the testing loss is of the same quantity order than the training one, the model well-learned the features and the classification is good. Otherwise, if the testing loss is too higher, there is a case of overfitting, which can be resolved by changing the features to extract or augmenting the training data.

2.3 Objects detection

The previous step provides the model weights that allow to best classify the three groups of point clouds considered. Now, we can use such results to detect some target objects sparsely distributed on the 3D space.
The idea is to slide over the 3D space, after splitting it into groups of 1000 points, and give each of these batches as input to the model, using the learned weights provided during the training; according to what the classifier outputs, the current batch is assigned to one of the three target clusters or no-one of them. This term takes the name of 'sliding window approach'.
Its main disadvantage is the huge computation cost and the consequent huge computation time.
A different way to perform this approach consists on the usage of the convolutional neural networks, which are going to be explained in details in the next chapter.

Chapter 3

Object detection model: Yolo

3.1 History and brief explanation

3.1.1 Classification and Regression

[6] It's possible to classify deep learning in two parts: classification and regression. Both of them has the task to extract feature and detect some objects, but the final goal is different.

- The classification problem maps the input function to a label, which consists on an integer number starting from 0 that identifies a category. For example, we want to discriminate dogs from cats; label 0 denotes dogs and label 1 denotes cats. The classification is obtained simply by adding a fully connected neural network at the end of the model, whose outputs are the labels.
- The regression problem is a bit more complicated; in this case, the goal is not only the classification, but also the localization of the object inside an environment (e.g. an image). The detection is performed through bounding boxes (parametrized with center and size), confidence scores and class probability scores. In this case, no classifiers are added at the end; the output of the model is a vector containing the bounding boxes, each one with its confidence score.

YOLO, the model we are going to describe, is a regression problem. The technique described in the previous chapter is a classification problem.

3.1.2 Explanation of Yolo

[7] Yolo (You Only Look Once) is a state-of-the-art detection algorithm, able to classify and localize up to 9000 objects in one image.

Such system is not based on classification, it's based on regression; this means that it's going to predict classes and bounding boxes for the whole image in only one run of the algorithm. Yolo's task is to predict a class of an object and localize it through a bounding box. In practice, it's going to predict 6 values: box center (x and y), box width, box height, box confidence and relative class (integer from 0 to n, where n corresponds to the number of classes).
In order to get more confidence with details of Yolo architecture, it's necessary to introduce some theoretical backgrounds about Neural Networks, in particular the CNN.

3.2 CNN and Max pooling: theoretical background

3.2.1 Neural Network

Definition [8] Artificial neural networks are one of the main tools used in machine learning. As the "neural" part of their name suggests, they are brain-inspired systems which are intended to replicate the way that humans learn. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.

Model [8] An Artificial Neural Network is composed by units called "neurons." The network consists of connections, each connection transferring the output of a neuron i to the input of a neuron j. In this sense i is the predecessor of j and j is the successor of i. Each connection is assigned a weight w_{ij}. Sometimes a bias term is added to the total weighted sum of inputs to serve as a threshold to shift the activation function.
Each neuron is modelized as follows.
Suppose to assign a label j to a neuron; it consists of the following components:

- input function $p_j(t)$, t corresponds to the (discrete) time;
- activation function $a_j(t)$: the neuron's state;
- output function $o_j(t)$ corresponding to the output of the activation function: $f_{out}(a_j(t))$

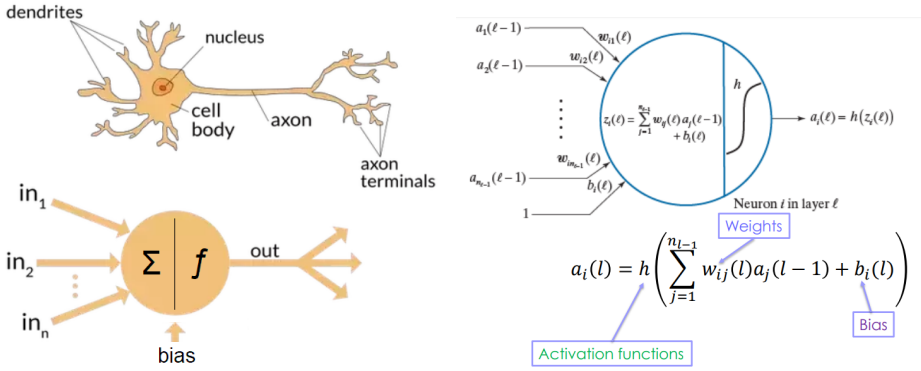


Figure 3.1: Artificial neural network

3.2.2 Fully connected neural network

[9] In a fully connected layer each neuron is connected to every neuron in the previous layer, and each connection has its own weight. This is a totally general purpose connection pattern and makes no assumptions about the features in the data. It's also very expensive in terms of memory (weights) and computation (connections).

NN Layers The simplest Neural Network is composed by three layers:

- an input layer: here are stored the (preprocessed) data we want to analyze (e.g. image pixels, function values, point coordinates etc); each node (neuron) of the layer stores one of each values.
- a hidden layer: each node receives the weighted sum of the input values; naming the hidden node output as l, weight matrix as W_1, biases vector as b_1, input function vector as x and activation function as $h(.)$, such layer is characterized by the following equation:

$$l = h(W_1 x + b_1)$$

- an output layer: each node receives the weighted sum of the outputs of the hidden layer. In case of a classification problem, this layer stores the confidence scores of each class; hence, the number of nodes is equal to the number of class. In case of a regression problem, considering the Yolo task, it stores the values of the bounding boxes that localize the object we want to detect; hence, it memorizes center, sizes, confidence and which class the object belongs to. Naming the weight matrix as W_2 and biases vector as b_2, such layer is characterized by the following equation:

$$y = = h(W_2 l + b_2)$$

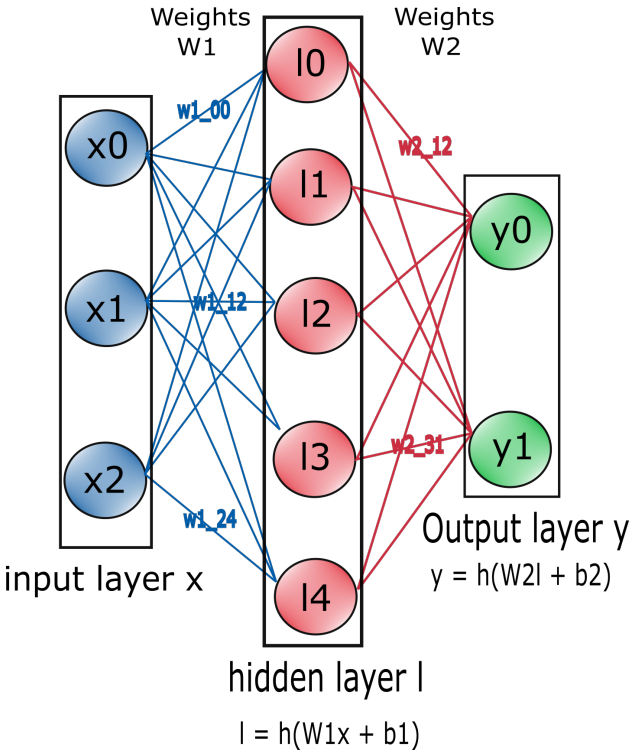


Figure 3.2: Fully connected neural network

Summarizing, here we have described the simplest scheme of a fully connected neural network; intuitively, the complexity of the network is directly proportional to the number of hidden layer (depth of the network) and the number of nodes in each one (width of the network).

Activation function [9] The activation function of a node defines the output of that node, or "neuron," given an input or set of inputs. It can be of several types:

- Rectified Linear Units (ReLU): this function maps from R^n to R^n_+. It guarantees a fast training thanks to its non-saturating nature (prevents gradient vanishing problem); it requires light computation. The equation is the following:

$$f(x) = max(0,x)$$

- Sigmoid function: maps from R^n to $[0,1]$, is characterized by a smooth output, represents the firing rate of a neuron. Since it has a saturation

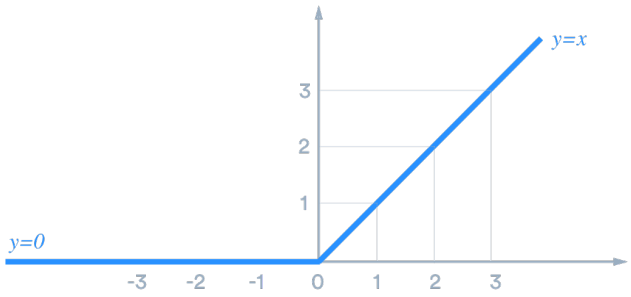


Figure 3.3: Rectified Linear Units (ReLU) activation function

nature, there is the possibility that gradient are killed around the saturation zones (near 0 or near 1) guaranteeing, in this way, a poor learning in this area. The equation, shown in Z transform (frequency domain), is the following:

$$h(z) = \frac{1}{1+e^{-z}}$$

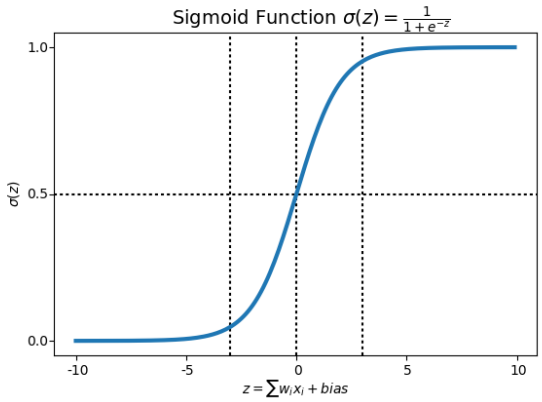


Figure 3.4: Sigmoid activation function

- Hyperbolic tangent (tanh): maps from R^n to $[-1,1]$, its saturation is similar to sigmoid and provides a zero-centered output. Hyperbolic tangent corresponds to a scaled sigmoid:

$$tanh(x) = 2sigm(2x) - 1$$

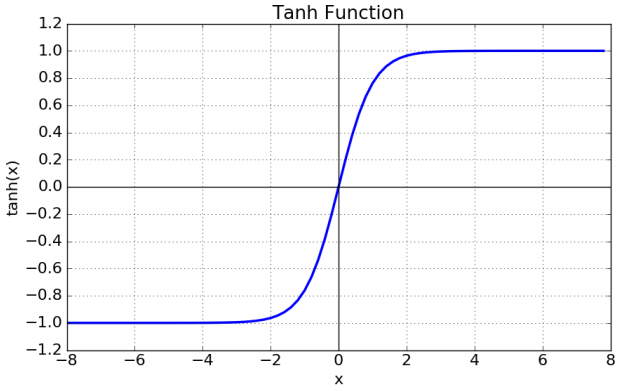


Figure 3.5: Hyperbolic tangent activation function

Forward and back propagation

Forward propagation [9] Forward propagation consists on mapping from input (x) to output (y) of the neural network. The input x provides the initial information that then propagates to the hidden units at each layer and finally produce the output y. Let's consider a deep fully connected neural network:

- Layer 1: $a_j(1) = x_j,\ j = 1, ..., n_1$
- Layer 2+: compute the value from the output of the previous layer:

$$a_i(l) \quad = \quad h\left(\Sigma_{j=1}^{n_{l-1}} w_{ij}(l)\, a_j(l-1) + b_i(l)\right)$$

- Network output: is the output of the last layer $a_j(L)$

Back propagation [3] It allows the information to go back from the cost backward through the network in order to compute the gradient. Let's consider the cost function E; in case of classification problem, such function corresponds to:

$$E \quad = \quad \frac{1}{2}\Sigma_{j=1}^{n_l}\left(r_j - a_j(L)\right)^2$$

where r_i is the vector containing the expected outcome, i.e. $r_i = 1$ in case of correct classification, 0 otherwise.

Consider now the neuron outputs z_j, let's analyze how cost depends on such value:

$$\delta_j(l) \quad = \quad \frac{\partial E}{\partial z_j(l)}$$

Starting from last layer (outcome) we obtain the following:

$$\delta_j(L) = \frac{\partial E}{\partial z_j(L)} = \frac{\partial E}{\partial a_j(L)} h'(z_j(L))$$

Now, backpropagate, namely compute all $\delta_j(L)$ and get $\delta_j(L-1)$, $\delta_j(L-2)$, ..., $\delta_j(1)$. This operation is done by exploiting the chain rule:

$$\delta_j(l) = h'(z_j(l)) \Sigma_i w_{ij}(l+1) \delta_i(l+1)$$

The derivatives with respect to weights and biases are:

$$\frac{\partial E}{\partial w_{ij}(l)} = a_j(l-1)\delta_i(l)$$

$$\frac{\partial E}{\partial b_i(l)} = \delta_i(l)$$

Finally, we update the parameters using the Gradient Descent:

$$w_{ij}(l) = w_{ij}(l) - \alpha \frac{\partial E}{\partial w_{ij}(l)} = w_{ij}(l) - \alpha \delta_i(l) a_j(l-1)$$

$$b_i(l) = b_i(l) - \alpha \frac{\partial E}{\partial b_i(l)} = b_i(l) - \alpha \delta_i(l)$$

where α denotes the learning rate.

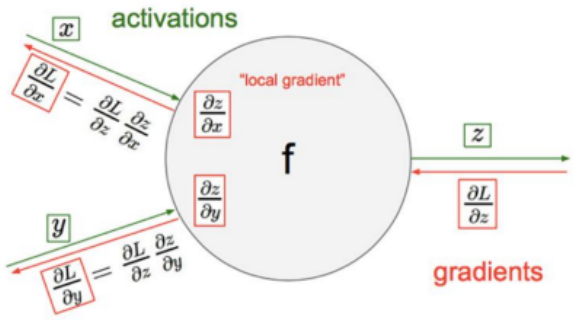


Figure 3.6: Local backpropagation scheme: explanation of what happens inside a node during the backpropagation process

3.2.3 Convolutional Neural Network

Definition [10][11] In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. It can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.

Differences from FCN [10][11] Comparing CNN with fully connected NN, we state that the main difference between them is the local connectivity. This means that a node of a convolutional layer is not connected to all nodes of the previous one but only to a window of nodes of a predefined size (usually $k = 3$). This connection is made through a convolutional operation with a kernel of dimension kxk, with k denoting the window size. Local connectivity entails a lower number of weights to train. The kernel shift along the image and with a predefined stride (usually 1).
Another important feature about CNN is that the weights are "shared". This means that every node of a layer shares the same weights with each other nodes of the same layer but operates on a different window.

Convolutional layer [10][11] Let's see in details what happens in CNN when the information propagates from a layer to the next one. Let's consider two layers: layer l and layer $l+1$. The input of a node of the $l+1$ layer corresponds to the result of an operation of convolution between a kxk kernel (matrix of weights) and a window of k nodes of layer l. In formulas, given:

- a kernel K of dimension $3x3$:

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

- a window of nodes of the same size of the kernel:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The input of next layer node is provided by the convolution:

$$\Sigma_{i=1}^{3}\Sigma_{j=1}^{3} a_{ij} k_{ij}$$

namely

$$A * K$$

The convolution allows to extract local features from images. For example, if we convolve a small image window with Sobel kernel, is obtained the local gradient, which allows to enlighten local image intensity variations like edges or corners. Such features make the difference in order to classify an object.

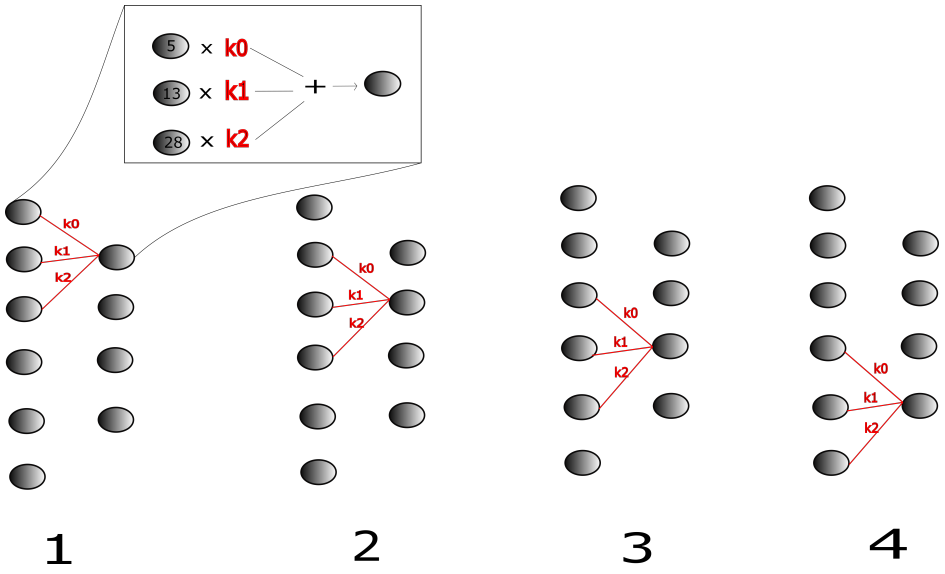


Figure 3.7: Convolutional neural network in 1D

Feature maps In CNNs, it is possible to learn multiple feature maps. A feature map correspond to a layer whose nodes inputs are calculated with a specific kernel of weights. If the kernel is different, we obtain another convolutional layer, namely another feature map. Intuitively, in case of images, a feature map encodes local image features, for example local curvature, linearity, intensity, edges or any type of variation. The more the network is deep, the more the features we extract are precise. Obviously, the choice of NN depth must be a good compromise between accuracy (deeper is better) and efficiency (less deep is better).
Yolo algorithm is designed to detect object of variable sizes; such sizes are clustered in three sets, which are going to define the anchors dimensions. For every sets, Yolo extract the correspondent feature map.

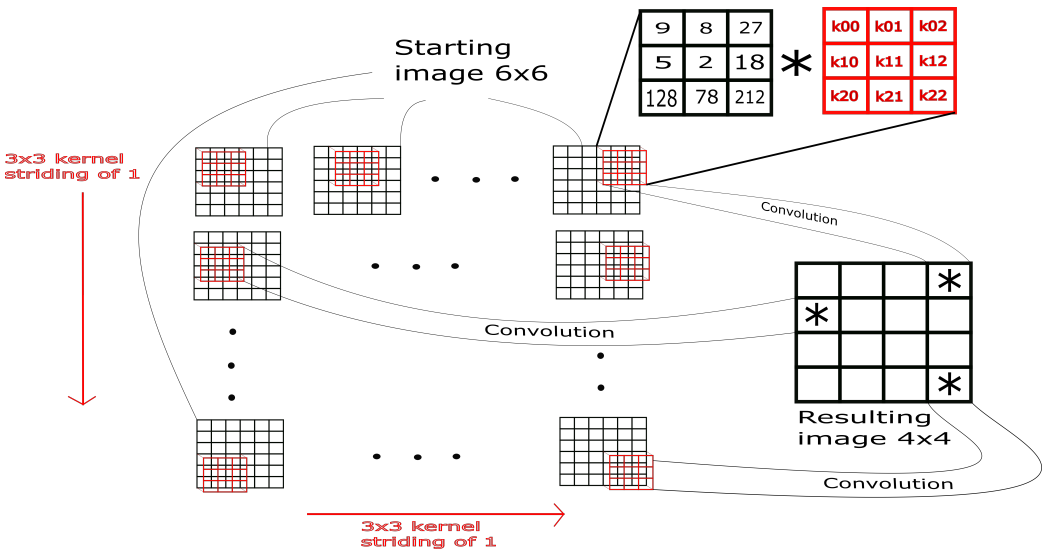


Figure 3.8: 2D convolutional neural network in 2D (images)

Padding [10] The convolution operation may lead to two type of results:

1. The output as greater or equal dimension than the input; if we input a $5x5$ image and we want to output the same size, we should perform a padding operation. Indeed, if we convolve the $5x5$ input with a $3x3$ kernel, we obtain a $3x3$ output matrix. To avoid this, it's performed a padding operation to the input, which consists on adding 0s in such a way to augment it from $5x5$ to $6x6$. In this way, once the convolution is applied, the output will be $5x5$. Since it will be of the same size of the input, the whole operation is named "same padding".

2. On the other hand, if the output has to be a lower dimension, we omit the padding operation, so obtaining an output of the same kernel size (in this case $3x3$). The whole operation is named "valid" padding.

Pooling [10] When a convolutional layer presents too many nodes, may be useful to perform a subsampling. This can be done by exploiting the pooling operation, which takes a small window of nodes and compute the maximum (max pooling, more used) or the average value of them (average pooling). In this way, the spatial size is considerably reduced, with a consequent increase of the efficiency.
The effects it can bring are:

- reduce the resolution, next layer can operate to a larger scale;

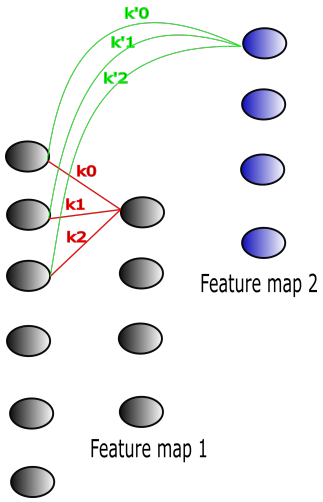


Figure 3.9: Feature maps in 1D

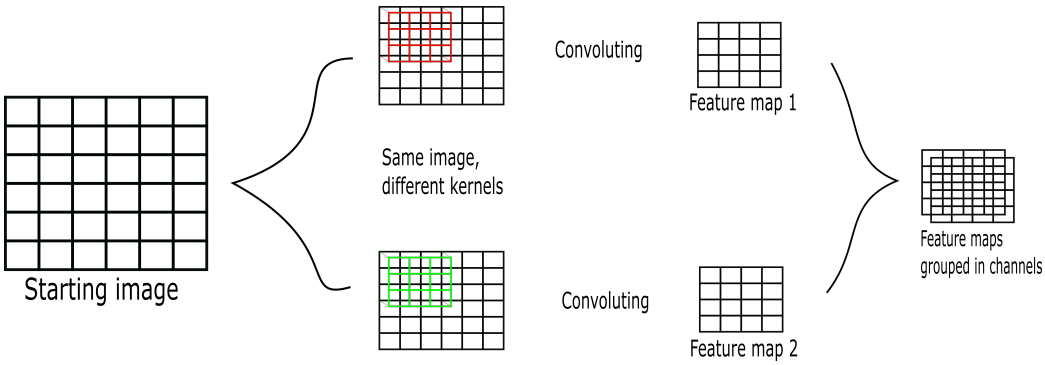


Figure 3.10: Feature maps in images

- reduce computational complexity;
- adding deformation invariance;
- max pooling is more used because it's faster and efficient.

Pooling is also useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model. Furthermore, max pooling is often used as noise suppressant: it discards the noisy activations altogether and also performs denoising along with dimensionality reduction.

In this part was summarized a theoretical background about Artificial Neural Networks, in particular the Convolutional Neural Networks (CNNs). The next

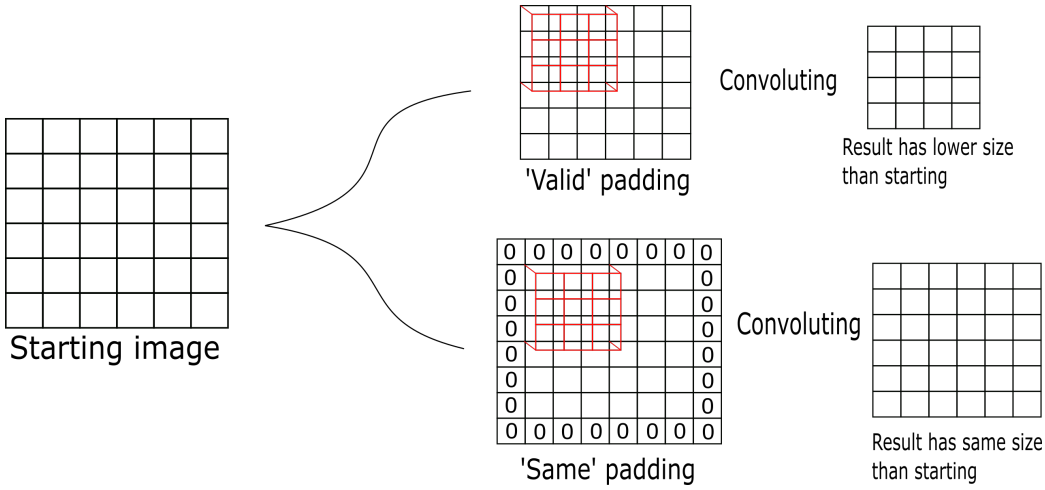


Figure 3.11: Padding

step is understanding how to train such powerful tools in order to best perform an object detection and classification.

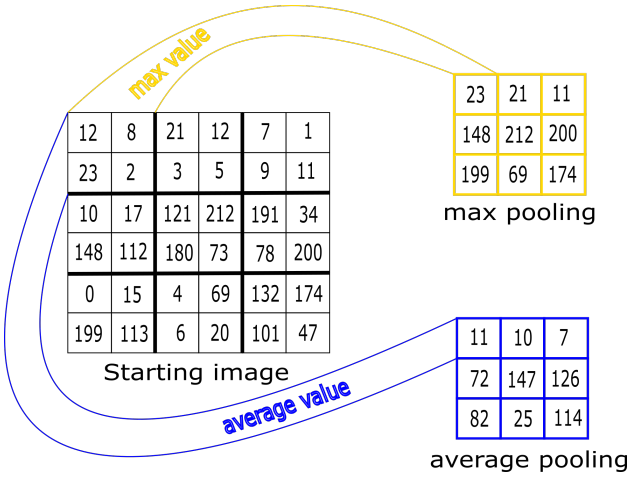


Figure 3.12: Max pooling and average pooling

3.2.4 Training and loss function

In order to achieve good detection and/or classification results, the network needs to be trained in order to make it learn the necessary features from the image. This operation, named "training", requires a huge number of training images. Such images feed the network during all the training process, also more

than once (n > 1 epochs); their task is to provide to the network, through an operation named "image labeling", as more information as possible. Let's see in details what training consists on. First of all, the training images need to be preprocessed.

Preprocessing and batch processing The preprocessing operation (also called "data augmentation") denotes all that process about preparing the images to feed the network. The goal is adding some variance to such data (noise, rotation, flipping). Indeed, if we want the network to classify dogs, we need to provide a large number of dog images. If such images are goodly preprocessed, the network will learn the typical dog features like tail, ears, head and body shape or legs; in case of bad preprocessing, the network could learn bad features like color (images were not grayscaled) or position (images were not rotated and flipped).

The images also need to be organized to avoid too slow computation and ex-

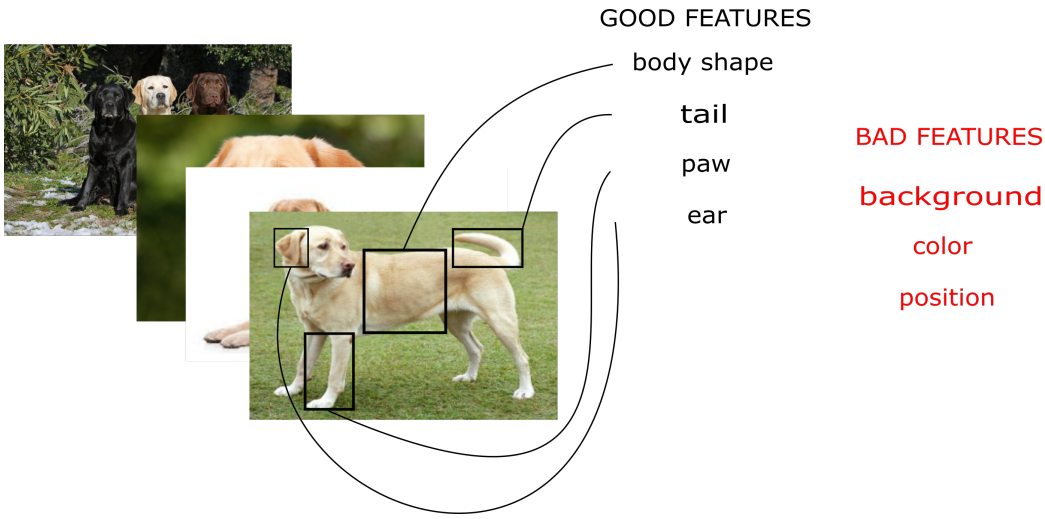


Figure 3.13: Training preprocessing: example of good and bad feature extraction

pensive memory usage; if we have 500 images, it's advisable to feed the network not with all the images a time, but in small groups (batches). The number of images per batch is called "batch size". Typical batch size values could be 8, 16 or 32; sometimes 64 too. Such batch arranging takes the name of "batch preprocessing".

Overfitting Overfitting occurs when the network does not learn general features but only features for a specific subset of target objects. For example, con-

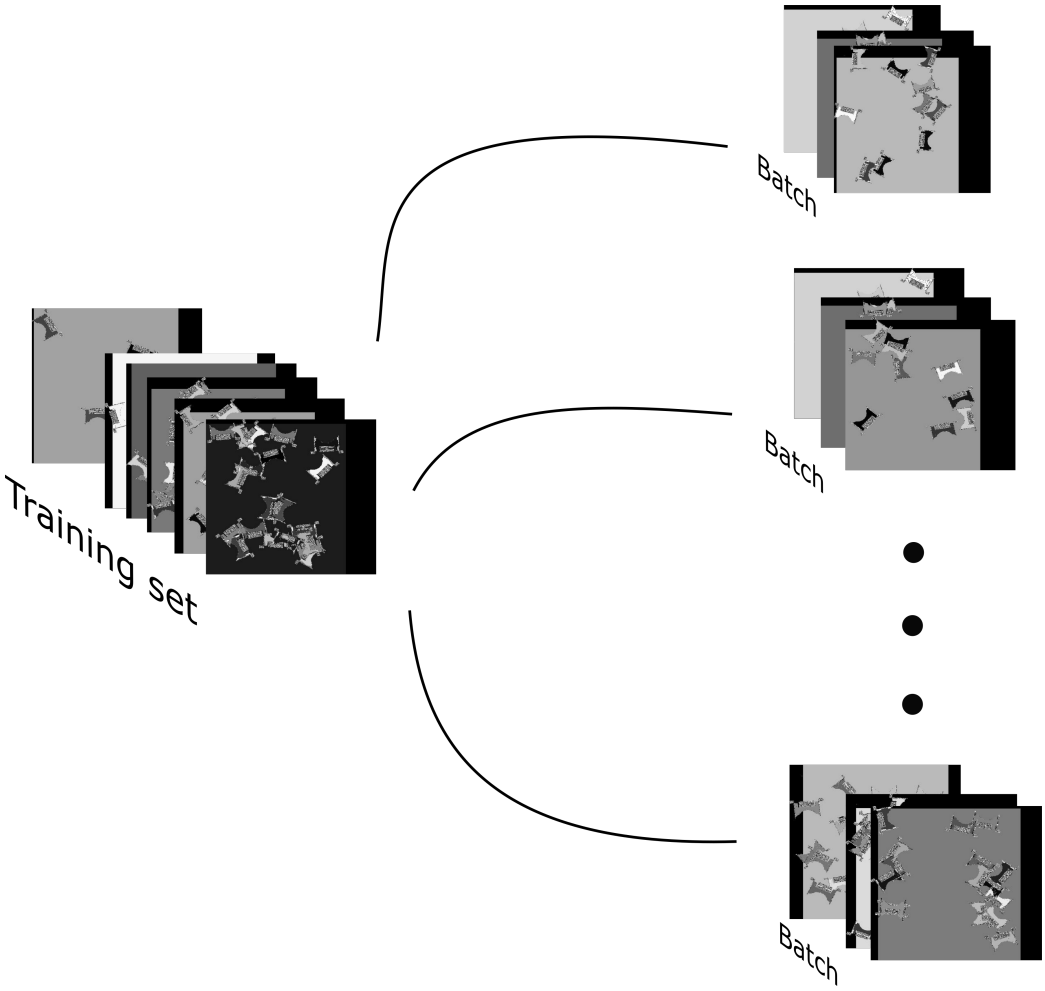


Figure 3.14: Batch preprocessing

sidering the above case of dogs, overfitting may due to bad feature learning such as color, positions; if training images contain too many black dogs, the network will probably learn to recognize dogs by their black color and discard all the other ones with different colors, with consequent low accuracy. This is an example of overfitting. It mainly occurs when the training images show too invariant features or simply when there are few training images.

Regularization In order to better stabilize the network, there are some regularization techniques:

- Dropout: units (nodes) randomly dropped during training (the ones whose value is under a certain threshold); units with value greater or equal to a

threshold (probability value) are retained. This method allows to thin the network by reducing the number of nodes, in such a way to improve computation and memory usage.

- L2 weight decay: big weights are penalized through an adjustable decay value; big weight decay value means high penalization for big weights.

- Early stopping: according to validation error, we decide when the training process should be stopped. For example, when the convergence is reached and the validation loss constantly lies under a certain value.

Optimization: Gradient descent, Adam, Momentum The training task is the minimization of the loss function. Three ways are going to be explained:

- [3]Gradient Descent Optimizer: divided in basic, stochastic and mini batch.

  - Basic: Computes the gradient of the cost function with respect to the parameters for the entire training dataset. It can be very slow and is intractable for datasets that don't fit in memory.
    It's a general approach for minimizing a differentiable convex function $f(w)$. The gradient $\nabla f(w)$ of $f(w)$ at $w = (w_1, \ldots, w_d)$ is:

    $$\nabla f(w) \;=\; \left(\frac{\partial f(w)}{\partial w_1}, \ldots, \frac{\partial f(w)}{\partial w_d}\right)$$

    The gradient points in the direction of the greatest rate of increase of f around w.

    Now, let α, $\alpha > 0$ be a parameter (learning rate). According to Basic Gradient Descent, during the backpropagation, weights are updated as it follows:

    $$w^{(t+1)} \;=\; w^{(t)} \;-\; \alpha \nabla f(w^{(t)}), \qquad t = 0, 1, \ldots, T-1$$

    If f is convex and ρ-Lipschitz continuous, convergence is always guarantees in a finite number of iterations.

  - [3]Stochastic: the idea is, instead of using exactly the gradient, taking a (random) vector with expected value equal to the gradient direction.
    Choose v_t at random from a distribution such that:

    $$E[v_t | w^t] \;\epsilon\; \nabla f(w^t)$$

The weights are updated as it follows:

$$w^{t+1} = w^t - \alpha v_t, \qquad t = 0, 1, \ldots, T-1$$

If f is convex and ρ-Lipschitz continuous, convergence is always guarantees in a finite number of iterations.

- Batch: is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated (after a training epoch). It may require a lot of memory usage.
  Instead of the whole dataset, the algorithm may update the model after a small set of the training examples; in this case, we are dealing with a mini-batch gradient descent optimizer. Less memory usage and faster computation.

- [4]Adam optimizer: also called "Adaptive Moment Estimation". It works through the following steps:

  1. Computes an exponentially weighted average of past gradients and stores it on variables V_{dW} and V_{db} (without bias correction), $V_{dW}^{corrected}$ and $V_{db}^{corrected}$ (with bias correction).
  2. Same operation, but with exponentially weighted average of the squared past gradients. The storing variables are S_{dW}, S_{db}, $S_{dW}^{corrected}$, $S_{db}^{corrected}$.
  3. Updates parameters in a direction based on combining information from 1 and 2.

  Adam optimization is implemented through the following steps:

  - Initialize V_{dW}, V_{db}, S_{dW}, S_{db} to zero.
  - On iteration t, compute the derivatives dw and db using the current mini-batch.
  - Update V_{dW} and V_{db}:

$$V_{dW} = \beta_1 V_{dw} + (1-\beta_1) dW$$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1) db$$

  - Update S_{dW} and S_{db}:

$$S_{dW} = \beta_2 S_{dw} + (1-\beta_2) dW^2$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

- Implement bias correction:

$$V_{dW}^{corrected} = \frac{V_{dW}}{1-\beta_1^t}$$
$$V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t}$$
$$S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2^t}$$
$$S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t}$$

- Update W and b:

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}+\epsilon}}$$
$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}+\epsilon}}$$

where:

- β_1 and β_2 are hyper parameters that control the two exponentially weighted averages. In practice we use the default values for $\beta_1 = 0.9$ and $\beta_1 = 0.999$.
- ϵ is a very small number to avoid dividing by zero ($\epsilon = 10^{-8}$).
- α is the (adaptive) learning rate.

- [5]Momentum optimizer: this technique is used along with GD. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go. As a reminder, the equation of Gradient Descent is the following:

$$w^{(t+1)} = w^{(t)} - \alpha \nabla f(w^{(t)}), \qquad t = 0, 1, \ldots, T-1$$

Such operation uses only the gradient of the current step to guide the search.
In momentum optimization, are also accumulated the gradients of the past steps to determine the direction to go:

$$v^{(t+1)} = \eta v^{(t)} - \alpha \nabla f(w^{(t)}), \qquad t = 0, 1, \ldots, T-1$$

with:

$$w^{t+1} = v^t + w^t$$

η denotes the coefficient of momentum, which consists on the percentage of the gradient retained at every iteration.
Notice that the weights increase for each update. As a consequence, decomposing the gradient into the two image dimensions, the momentum allow to dampen the zig-zag oscillation that occurs during the (stochastic) gradient descent optimization; in this way, the first dimension tends to be canceled out and the second one is reinforced with a obvious consequent quickening convergence to minimum.

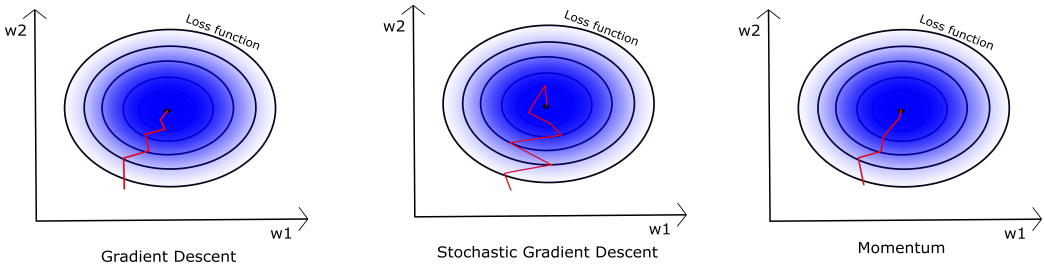


Figure 3.15: Optimization algorithms description

3.3 Yolo model

At this point, after a theoretical background of the basic deep learning tools, neural networks, let's go to describe the deep-learning based algorithm that will be used for image detection: YOLO.

3.3.1 Preprocessing: from point cloud to image

Before feeding YOLO, it's needed to perform some data preprocessing. Since the starting data is not an image, but a point cloud, we are going to describe all the steps to transform it to a suitable input for the system.

Point cloud adjusting

The original detected point cloud needs to be adjusted before being converted to image. The detection, indeed, shows many outliers and results not be aligned with the dimensions of the world reference system.

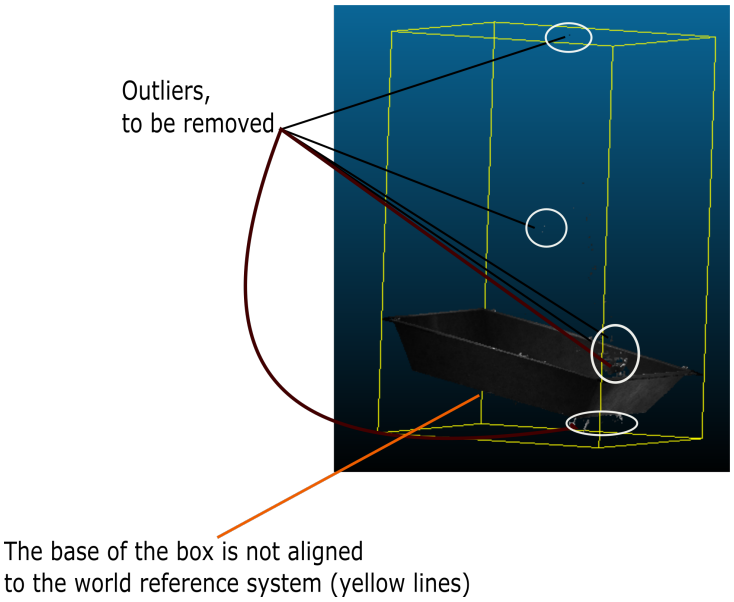


Figure 3.16: Original acquired point cloud

The outliers removal is simple: it's enough to scan all the (normalized) cloud points through a for loop and remove the ones that are out of a predefined range; such range, in this case, consists on the interval between minimum and maximum planes of the box
The second step requires an alignment of the box with the world axis, in order to have the base of the box lying on a plane with z-coordinate be more or less constant. To solve this problem, we resort to Principal Components Analysis, so eigenvectors of the point cloud covariance matrix. Such eigenvectors, indicate the directions of the point cloud along the world space; the goal is to rotated them in order to be aligned with the world canonical reference system.
Let's describe in details all the steps:

1. Compute the point cloud covariance matrix:

$$\Sigma = \frac{1}{N}\begin{bmatrix} (X-\mu_x)^T(X-\mu_x) & (X-\mu_x)^T(Y-\mu_y) & (X-\mu_x)^T(Z-\mu_z) \\ (Y-\mu_y)^T(X-\mu_x) & (Y-\mu_y)^T(Y-\mu_y) & (Y-\mu_y)^T(Z-\mu_z) \\ (Z-\mu_z)^T(X-\mu_x) & (Z-\mu_z)^T(Y-\mu_y) & (Z-\mu_z)^T(Z-\mu_z) \end{bmatrix}$$

where:

- X, Y, Z are the vectors storing, respectively, all the x, y, z coordinates of the point cloud;

- μ_x, μ_y, μ_z are, respectively, the average values of x, y, z coordinates of the point cloud;
- N is the number of points.

2. Extract the eigenvectors v_0, v_1, v_2, indicating the directions along, respectively, axis x, y, z of the world reference system. The vectors are then sorted according to correspondent eigenvalues, from largest to lowest, in such a way to obtain the directions stored in order of importance.

3. Consider now the third eigenvector, namely the one denoting the minor direction. The goal is using such vector to transform the point cloud in such a way to make the box base on a horizontal plane (i.e. with constant z); v_2 indeed denotes the normal of target plane. To achieve this, we proceeded in the following way:

   - align the eigenvector with canonical axes $e_1 = [0,1,0]^T$ and apply such transformation to all the points of the cloud; the steps are the following:
     (a) Obtain the rotation axes $k = [k_0, k_1, k_2]^T$, namely the cross product between e_2 and eigenvector v_2:

     $$k = e_2 \times v_2$$

     (b) Get the angle of rotation (in the form of its sine s and cosine c):

     $$c = e_2 \bullet v_2$$

     $$s = \sqrt{1 - c^2}$$

     Compute the point cloud transformation by multiplying it with the following rotation matrix:

     $$R = \begin{bmatrix} c + k_0^2(1-c) & k_0 k_1 (1-c) - k_2 s & k_0 k_2 (1-c) + k_1 s \\ k_1 k_0 (1-c) + k_2 s & c + k_1^2 (1-c) & k_1 k_2 (1-c) - k_0 s \\ k_2 k_0 (1-c) - k_1 s & k_2 k_1 (1-c) + k_0 s & c + k_2^2 (1-c) \end{bmatrix} \tag{3.1}$$

     At this point, the box base normal results to be aligned with canonical axes e_2.
   - rotate the resulting point cloud of -90 degrees around x-axes. Furthermore, after such rotation the box shows a light rotation of about -8 degrees around z-axes. It's enough to compensate by performing

a counterclockwise rotation of 8 degrees around z.
In formulas, denoting with $\theta_x, \theta_y, \theta_z$ the rotation angles around, respectively, x, y, z and denoting $c_x = \cos(\theta_x)$, $c_y = \cos(\theta_y)$, $c_z = \cos(\theta_z)$, $s_x = \sin(\theta_x)$, $s_y = \sin(\theta_y)$, $s_z = \sin(\theta_z)$, the following rotation matrices are computed:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & -s_x \\ 0 & s_x & c_x \end{bmatrix} \qquad R_y = \begin{bmatrix} c_y & 0 & s_y \\ 0 & 1 & 0 \\ -s_y & 0 & c_y \end{bmatrix} \qquad R_z = \begin{bmatrix} c_z & -s_z & 0 \\ s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The final rotation matrix corresponds to:

$$R = R_z R_y R_x$$

The angles used are $\theta_x = -90, \theta_y = 0, \theta_z = 8$

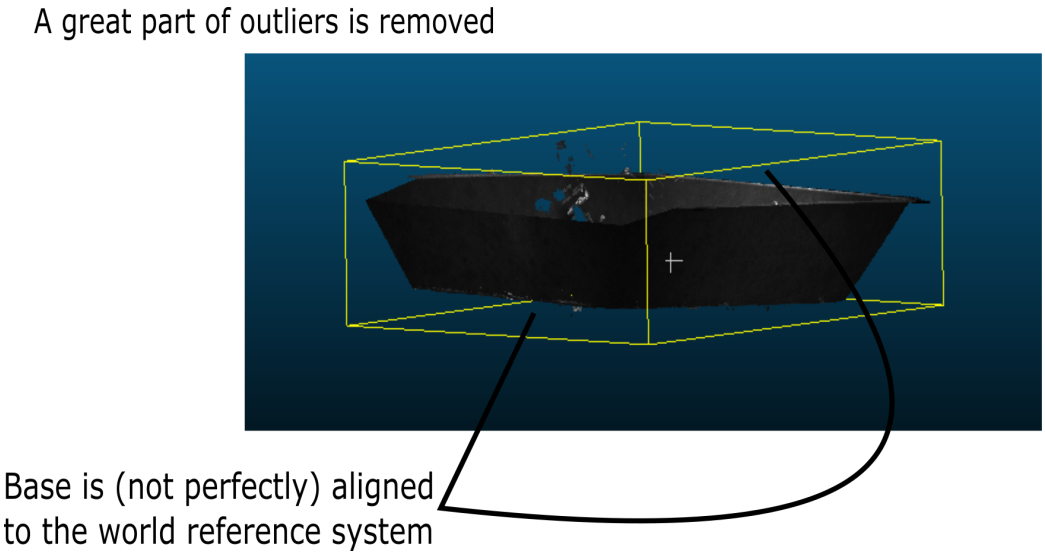


Figure 3.17: Adjusted point cloud

Conversion to image

Once the point cloud is adjusted, the next step consists on its conversion to a 2D image, in order to feed YOLO algorithm. The procedure is composed by two steps:

- point cloud voxelization
- image obtaining

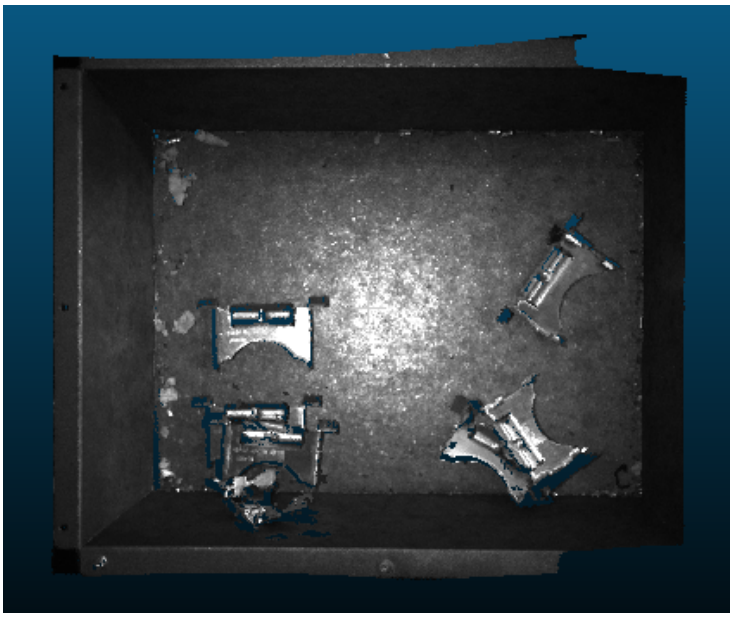


Figure 3.18: Top view of the acquired (and adjusted) point cloud

Point cloud voxelization This operation is crucial in order to obtain the image. It consists on organizing the sparse points of the cloud into a grid of 2D cells. The cells dimensions is equal to the box width and height (i.e. about 725 cm and 615 cm).
In details, each point of the cloud is splitted into its space coordinates x, y and z; the goal is to assign each of these points to the correspondent index of the voxel grid. Each x and y values is so used as a hash key of a hash function, which is now going to be explained. Let's denote x_s, y_s and z_s as the arrays storing all the x, y, z coordinates of the point cloud, x_{min}, y_{min} and z_{min} the x, y, z minimum values, x_{vox} and y_{vox} the number of cell along x direction and y direction; the hash function is the following:

$$x_s^{grid} = (x_s - x_{min}) \frac{x_{vox} - 1}{xyz_{max}}$$

$$y_s^{grid} = (y_s - y_{min}) \frac{y_{vox} - 1}{xyz_{max}}$$

where:

- x_s^{grid} and y_s^{grid} are the arrays storing all the grid indexes assigned to each cloud point;
- xyz_{max} corresponds to the (translated) point cloud maximum value:

$$xyz_{max} = max\big(max(x_s - x_{min}),\ max(y_s - y_{min}),\ max(z_s - z_{min})\big)$$

The hash function is not bijective: for each grid cell (indexed by a value from x_s^{grid} and one from y_s^{grid}) could correspond more than one point of the cloud. For the final goal, we need one point per cell; to solve this issue, for every point of each grid cell, it is considered only the one with maximum z value, namely the highest point for such cell. The final grid is so obtained; it corresponds to an 3D array with dimension $x_{vox} \times y_{vox} \times 1$. Thanks to hash table method to assign the indexes, each assignment is a O(1) operation, that means that all the voxelization is O(N), where N is the number of points of the cloud.

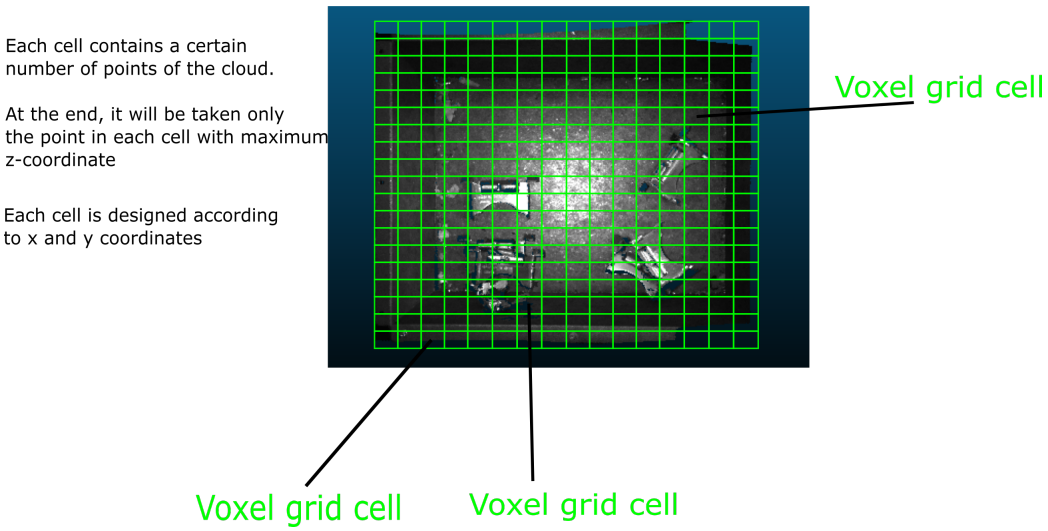


Figure 3.19: Point cloud voxelization

Image obtaining At this point, it's enough to convert the obtained voxels grid to an image format. Since the origin of image reference system (top left) is different from the point cloud one (bottom left), before conversion it's required to rotate the grid of +90 degrees (of course around z axes).
Finally, exploiting the suitable function provided by Pillow python library, it's possible to obtain the image and feed YOLO.

Let's explain now the architecture of YOLO. The model architecture is based on Darknet-53.

3.3.2 Darknet-53

[2] Darknet-53 is the name of the deep learning model which allows YOLO to extract features from the input image in order to best perform the detection. Its architecture is based on the Darknet-53 block.

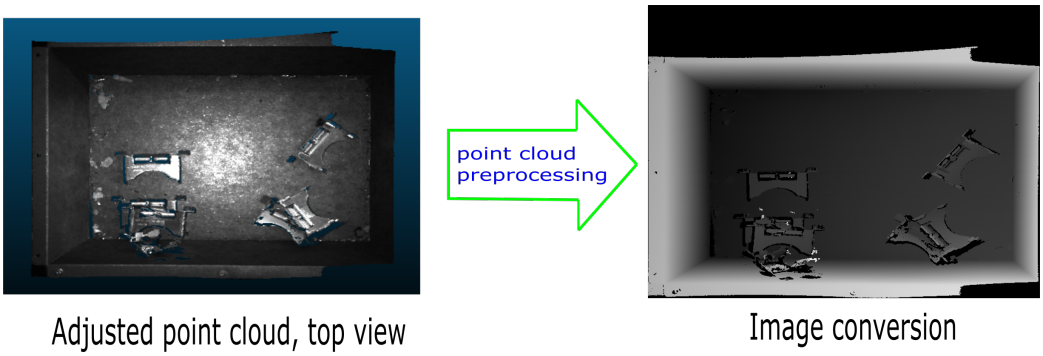


Figure 3.20: Obtaining of the point cloud image which is going to feed the deep learning model

Darknet-53 block

[2] It consists on two 2D convolutional neural networks; depending on kernel strides, the input image could previously padded ("same" padding) or not ("valid" padding). Both CNNs have 3×3 kernel size striding of 1.

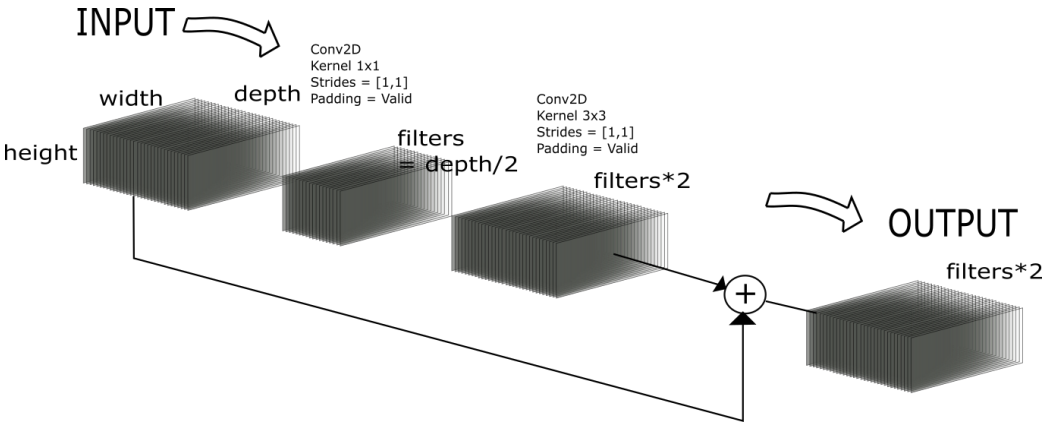


Figure 3.21: Darknet53 block feature extraction

Forward layer

[2] Such layer is the core of Darknet-53. It's composed by a sequence of convolutional neural networks with different strides, kernel sizes and output filters. The sequence consists on 2 CNNs followed by a darknet-53 block; then the model contains an alternation of a CNN with padding and sequenced of darknet-53 blocks. The total number of CNNs exploited corresponds to, as the name of the model suggests, 53. Such model provides three outputs:

- $route_1$: the output layer after the first sequence of darknet-53 blocks;
- $route_2$: the output layer after the second sequence of darknet-53 blocks;
- output: the final output layer after all the sequences of darknet-53 blocks.

Such outputs will be used for the three feature maps computation in forward layer of YOLO.

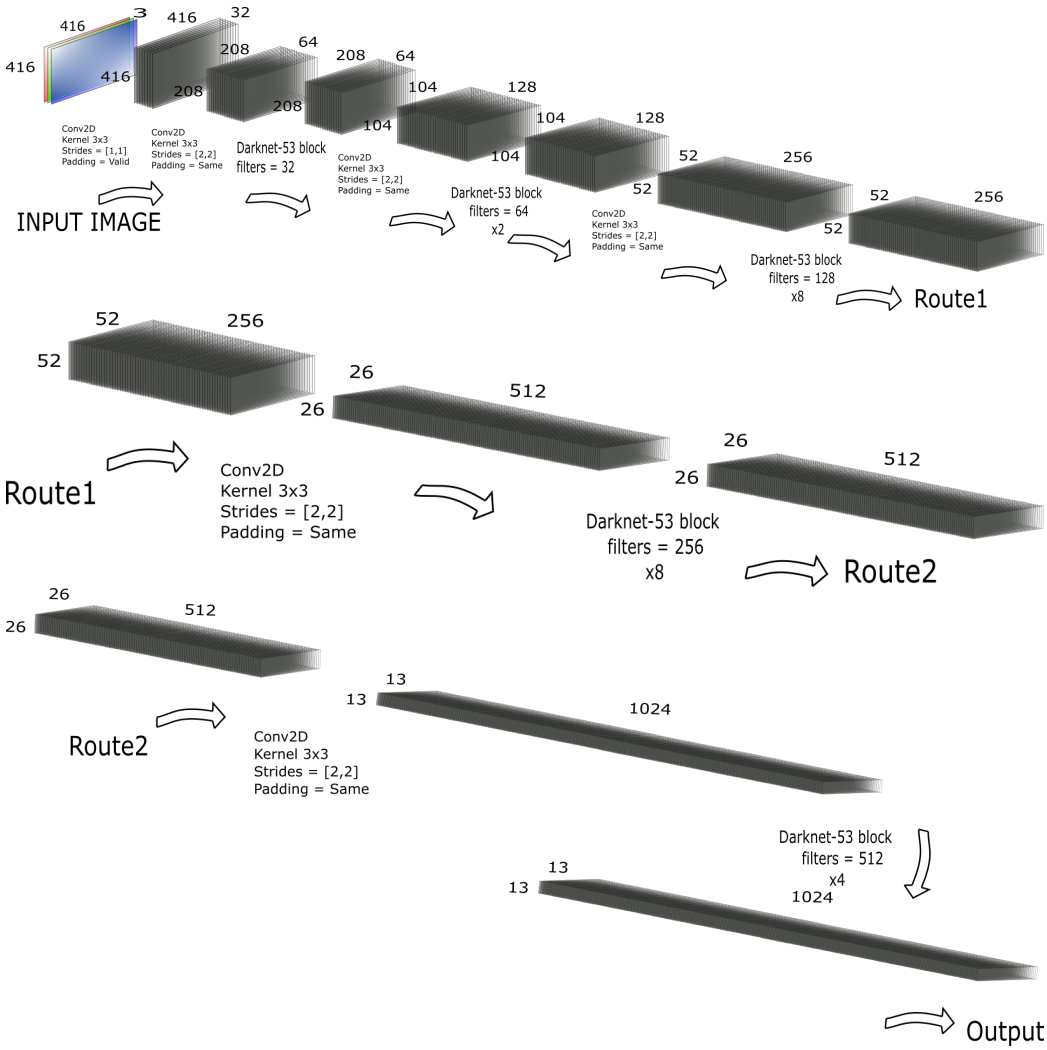


Figure 3.22: Three blocks of features extracted from Darknet53: route1, route2, outputs

3.3.3 Yolo

Darknet model is going to be used by YOLO algorithm for the image feature extraction and, finally, the bounding box detection. Let's go to see in details all the parts of this powerful tool.

Yolo block

[2] Yolo-block consists of a sequence of 6 convolutional neural networks, with alternated kernel size equal to 1×1 or 3×3, always striding of 1 unit along the image grid. It provides 2 outputs:

- route: the output layer of the first 5 convolutional neural networks;
- outputs: the final output layer of sequence of all 6 neural networks.

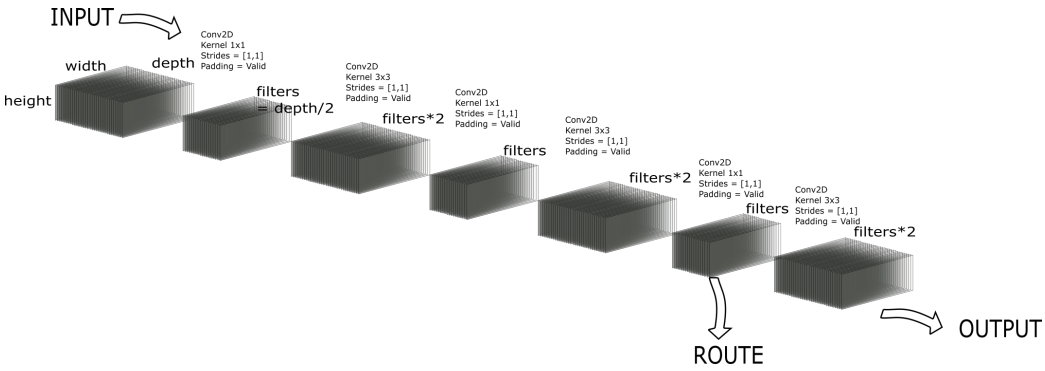


Figure 3.23: Yolo-block feature extraction

Forward layer

[2] To start, such layer takes the output of the darknet-53 forward: route_1, route_2 and outputs. It uses the latter as input to the yolo-block to compute the two values: route and outputs. 'Outputs' tensor is given as input to the detection layer in order to compute the first feature map (the one relative to 3 of the 9 anchors provided).
At this point, the $route$ tensor is given as input to another CNN, whose output (upsampled to $route_2$ size) is concatenated to $route_2$ variable. The concatenation result is given as input to yolo-block in order to compute the second feature map (relative to other 3 of the 9 anchors provided).
The same operations are performed to obtained the last feature map.

Anchors

The boxes parameters are not predicted as distance from the origin of the image (top-left). They are predicted with respect to prior predefined boxes named "anchors". The anchors are predefined only in terms of sizes, because their centers simply correspond to the center of each grid cell.

Re-organization layer

[2] This layer takes as input the feature map computed by the forward and detection layers. Its task consists on splitting the feature map tensor in order to get the detected bounding box coordinates, sizes, confidence and class scores. The feature map tensor stores $[t_x, t_y, t_w, t_h, c, p_1, \ldots, p_n]$, where:

- t_x, t_y are the box centers;
- t_w, t_h are the box sizes;
- c denotes the confidence score, namely the probability there is a detected object;
- $p_1, \ldots, p_n$ denotes the class scores, n is the number of classes.

The bounding boxes center coordinates are provided as distances from the top-left of the correspondent cell; so, it's needed to add them an offset.

$$c_x = Sigmoid(t_x) + offset_x$$

$$c_y = Sigmoid(t_y) + offset_y$$

where $offset = [offset_x, offset_y]$ denotes the distance between top-left of the cell and top-left of the image. The sizes are provided w.r.t anchor sizes, for stability reasons. The final ones are computed as it follows:

$$w_{final} = e^{t_w} w_{anchor}$$

$$h_{final} = e^{t_h} h_{anchor}$$

Confidence and class scores are computed as it follows:

$$c_{final} = Sigmoid(c)$$

$$p_{i_{final}} = Sigmoid(p_i) \qquad i = 1, \ldots, n$$

Detection layer

[2] Detection layer is the final convolutional layer, which provides the bounding boxes coordinates tensor $[t_x, t_y, t_w, t_h, c, p_1, \ldots, p_n]$.

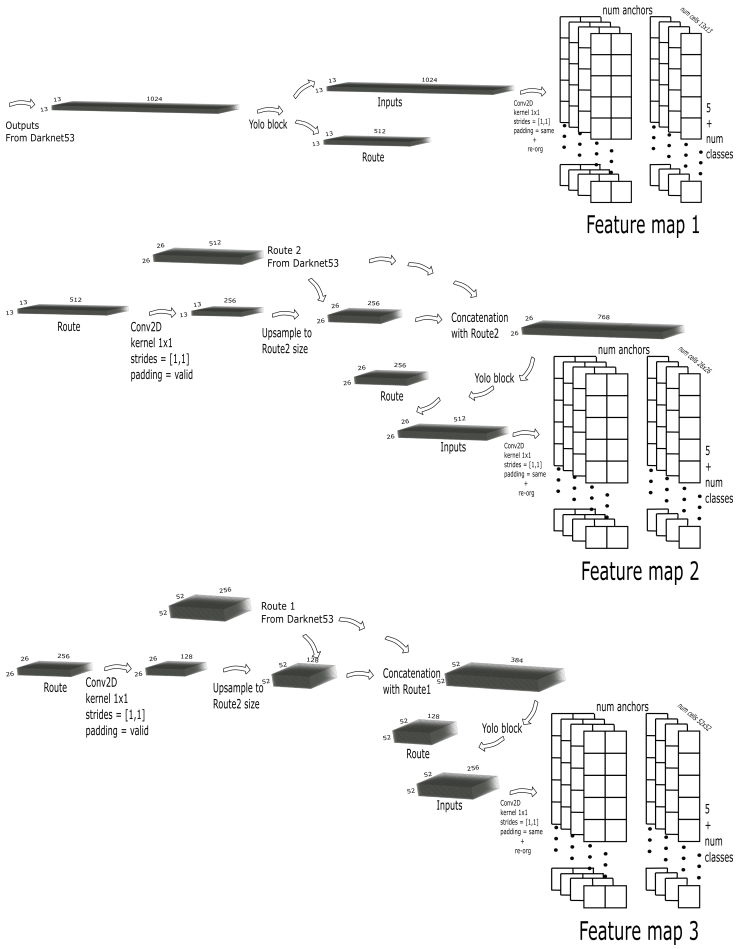


Figure 3.24: Three feature maps extracted from the YOLO forward + re-org + detection layer

Prediction layer

[2] Prediction layer takes the bounding boxes coordinates, confidences and class scores and arranges them in such a way to obtain: boxes tensor, confidence tensor and class probability tensor. Boxes centers coordinates are modified to obtain boxes vertexes coordinates. Hence, the function outputs such three tensors.

3.4 Yolo loss

[2] A loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost"

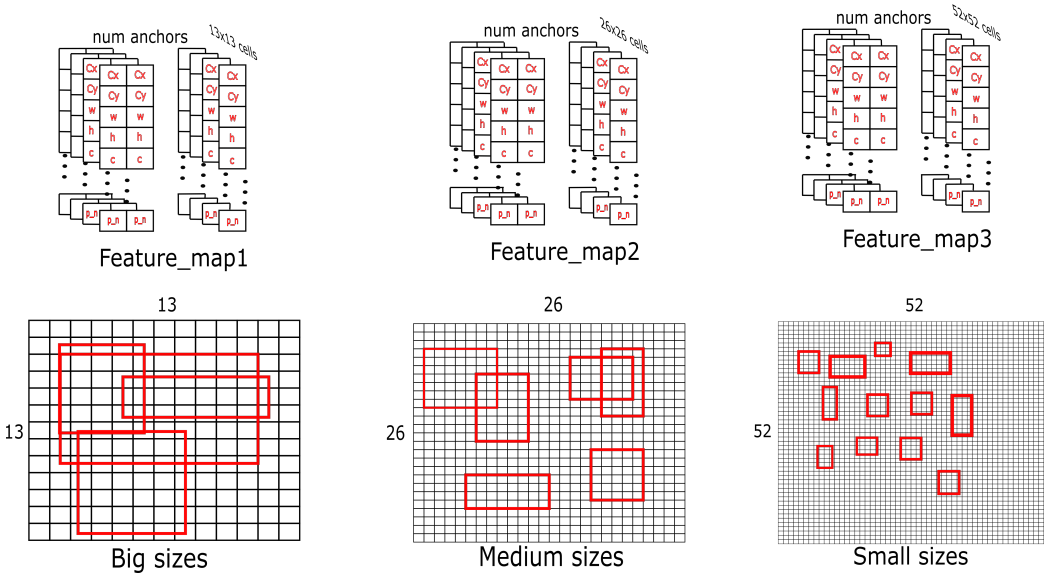


Figure 3.25: Feature maps to final bounding boxes

associated with the event. The goal of the optimization problem is the minimization of such value. For YOLO problem, the loss formulation is the following:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[(c_{x_i} - \hat{c}_{x_i})^2 + (c_{y_i} - \hat{c}_{y_i})^2 \right] \; +$$

$$+ \; \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[(w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2 \right] \; +$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[(c_i - \hat{c}_i)^2 \right] + \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \left[(c_i - \hat{c}_i)^2 \right] \; +$$

$$+ \sum_{i=0}^{S^2} 1_{i}^{obj} \sum_{cc classes} \left(p_i(c) - \hat{p}_i(c) \right)^2$$

Let's go to explain each term.

First term The first term encodes the squared error of the boxes centers coordinates; $\hat{c}_x$, $\hat{c}_y$ indicate the values of the feature map predicted by re-organization layer, c_x, c_y denote the value of the label boxes (ground truth boxes).
λ_{coord} is a scalar value used to give more or less weight to the loss term comparing it to the other ones. Such value depends on box and image sizes and it's

equal to:

$$\lambda_{coord} = 2 - \frac{w}{w_{im}} \frac{h}{h_{im}}$$

1_{ij}^{obj} is a boolean mask which denotes that the sum is going to consider only the coordinates of the predicted boxes which have an intersection over union (iou) with a ground truth box (obviously of the same class) greater than a predefined threshold (usually 0.5). In practice, if the iou is greater than threshold, inside that cell there is an object to detect; if not, there isn't an object, so the error simply goes to zero.
S^2 denotes the number of cells in the image grid, B denotes the number of predicted boxes inside each cell.

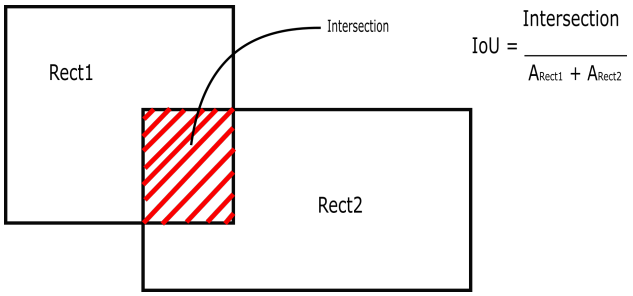


Figure 3.26: Intersection over Union

Second term It's exactly the same term as the first one. The only difference consists on the fact that in this case, instead of boxes centers, there are boxes sizes (predictions and ground truths).

Third term Such term is splitted into two parts and computes the squared error of the confidence scores.
The first part computes the confidence error in case the cell contains a target object. The ground truth value for confidence score is obviously equal to 1. The second part does the same in case there isn't any object to detect. In the latter case, ground truth confidence is equal to 0.
1_{ij}^{noobj} is the negative boolean mask of 1_{ij}^{obj}.

Fourth term The last term computes the class probability error. 1_{i}^{obj} is a boolean mask that denotes if the cell contains an object (= 1) or not (= 0). The metric is yet the intersection over union between predictions and ground truths.

The probability scores (both predictions and ground truths) are booleans one-hot encoded, namely only one of the n classes corresponds to 1, 0 the other ones. Such term simply denotes the classification loss.

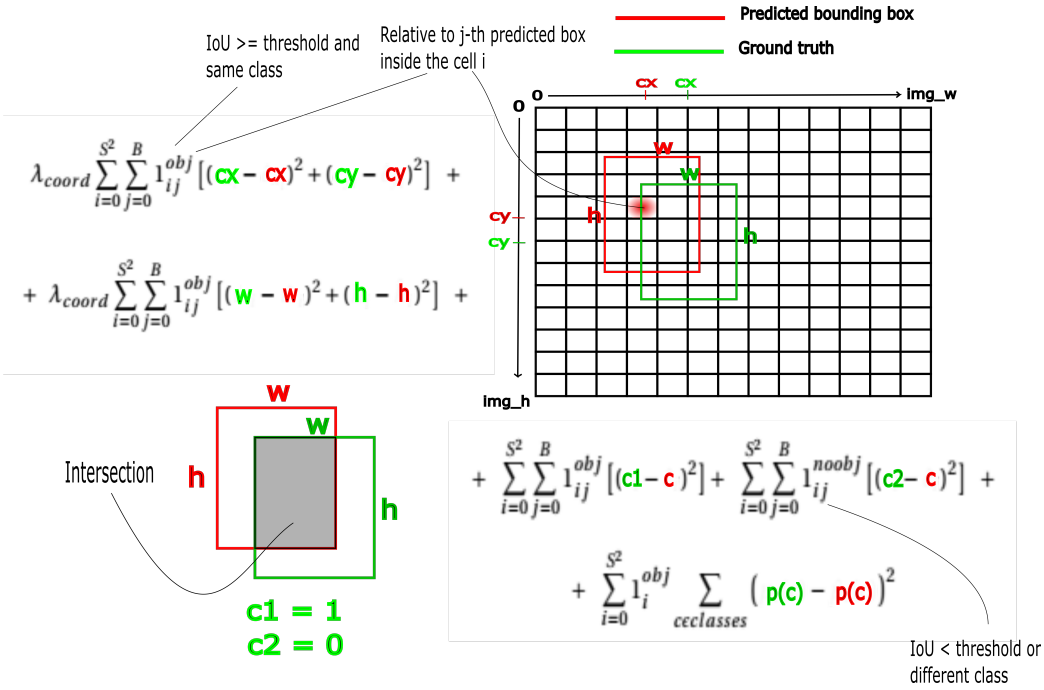


Figure 3.27: Loss function terms explanation

Chapter 4

Training and Validation

This chapter is going to explain in details all the steps of training and validation processes. In particular, we'll focus on two macro-steps:

1. Dataset preprocessing: consists on the generation and preparation of the training data; this step occurs before the actual training process.

2. Training process: consists on periodically feeding the YOLO network with the preprocessed training data; such process may last hours and it serves to find the right combination of parameters (weights) that minimize the loss function.

After the training process, a validation process is performed. Such operation consists on feeding YOLO with new fresh data, different from the training ones, in order to verify if the model learned well or enough to detect the target objects. The validation data must be of the same type of the training ones but with target objects distributed in a different way into the images (to avoid overfitting). Also the preprocessing must be the same.

4.1 Training

Let's go now to explain the first, very important process for training: the preprocessing.

4.1.1 Training images generation

The training data we want to generate contains a variable number of target objects randomly distributed and rotated along the image. To start, we have only a computer-made point cloud model of the target object. It's so necessary a function which generate the images starting from such model.

Point cloud data generation There are two steps to do in order to generate a good training point cloud:

- background generation;
- target objects generation.

The generation of the background is quite simple. It consists on a bi-dimensional grid of uniformly distributed points, its size is the same of the objects box one (about 725×615). The z coordinates of such points are constant along the grid and vary in a predefined interval.
The objects generation is a bit more difficult. The steps to do are:

1. object point cloud point-sampling;
2. object rotation and translation;
3. obtaining the final point cloud in top view.

The starting point cloud of the object contains about 3 millions points, too much if we want to make the process fast. It's needed to make a random sampling to obtain less points; the final number of points must not be neither too low nor too high. It needs to be large enough to obtain a uniform distribution of points when joining to the background. The chosen number is $140 \times 80 \times 10$; those three numbers represent more or less width, length and height of the target object.
At this point, the object needs to be rotated and moved, obviously staying inside the background grid size intervals. The rotation is performed by exploiting the same rotation matrices (R_x, R_y, R_z) explained in the previous sections; the angle θ_x and θ_y are null, while θ_z is a random number inside interval $[0,360]$. The rotation is performed around the center point of the object. The translation consists on adding to the object cloud a 3D vector $[t_x, t_z, t_z]^T$; t_x is a random float number that lies in the interval $[0,650]$, t_y in $[0,550]$ and t_z in $[0,25]$.
The object generation, rotation and translation are performed a random number of times, in the interval $[0,20]$.
At this point, we have a point cloud that shows:

- A set of 725×615 points composing the background;
- A set of points composing the objects (the number of object is random, up to 20); the x and y object coordinates lies inside the background grid interval; the zs (height), lies lightly over the background.

In order to get a faithful simulation of the reality (objects in a box seen in top view), the point cloud needs to be modified again in such a way to eliminate all the points that we cannot see from the top. We perform such top view generation through the following steps:

1. create a 2D grid of the same size of the point cloud one;

2. through an opportune hash function, assign each point of the cloud to the correspondent cell of the grid, according to its x and y coordinates; the hash function used is:

$$x_i^{grid} = int(x_i), \qquad i = 1,\dots,N$$

$$y_i^{grid} = int(y_i), \qquad i = 1,\dots,N$$

   where: N is the point cloud cardinality,
   x_i^{grid} and y_i^{grid} are the grid cell coordinates assigned to the i-th point,
   x_i and y_i are the coordinates of the i-th point;

3. for each grid cell, save only the point with maximum z (height).

Thanks to the hash function, each assignment speed is O(1), which makes the all function be O(N).
The final point cloud in top view is obtained by reshaping the grid (which contains one point per cell) from size $725 \times 615 \times 3$ to $725 \bullet 615 \times 3$.

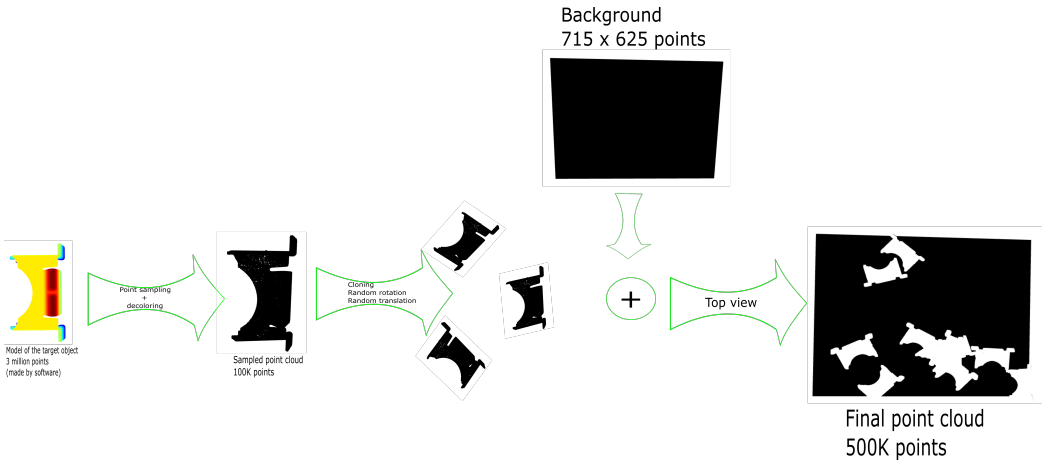


Figure 4.1: Point cloud data generation

Conversion to image At this point, we get a top view point cloud with a random number of target objects lying over a background. The next step consists on obtaining the final training image. The procedure is exactly the same explained previously for the image preprocessing: it consists on a point cloud voxelization (exactly the same function) and the exploiting of python 'Pillow' library to obtain the *RGB* image.

The whole operation is repeated for each training and testing image generates; the training images generated are 512, the testing images are 20.

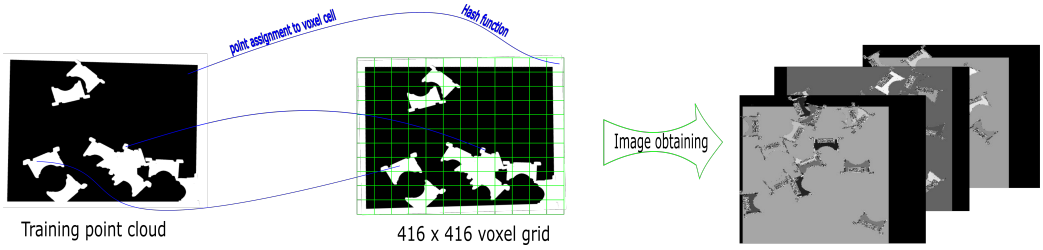


Figure 4.2: Conversion to image

4.1.2 Dataset preprocessing

The preprocessing operation is performed on the training images, namely the ones whose target objects are already been labeled (provided bounding boxes coordinates) from the outside. Each label is provided as:

- vertexes coordinates, in pixels. Each coordinate denote the distance (i.e. the number of pixels) from the top-left of the image;
- confidence score of the bounded object; in true boxes is obviously always equal to 1;
- Class probabilities scores, encoded in one-hot, denoting which class the bounded object belongs to (in case of only 1 class, such value is always 0).

In addition the true boxes, stored in tensors, are memorized according to three categories. Each of this groups is relative to each group of anchors utilized for the prediction; there are three groups of anchors, according to their size: small anchors, medium anchors and big anchors. The variable size is correlated to the size of the object we want to detect to. The true boxes categories are:

- $y_{true}13$: the 13 means that the image grid in which we are going to make the prediction is composed of 13×13 cells; the target objects here have a big size.

- $y_{true}26$: image grid 26×26 cells, for medium object sizes;
- $y_{true}52$: image grid 52×52 cells, for small object sizes;

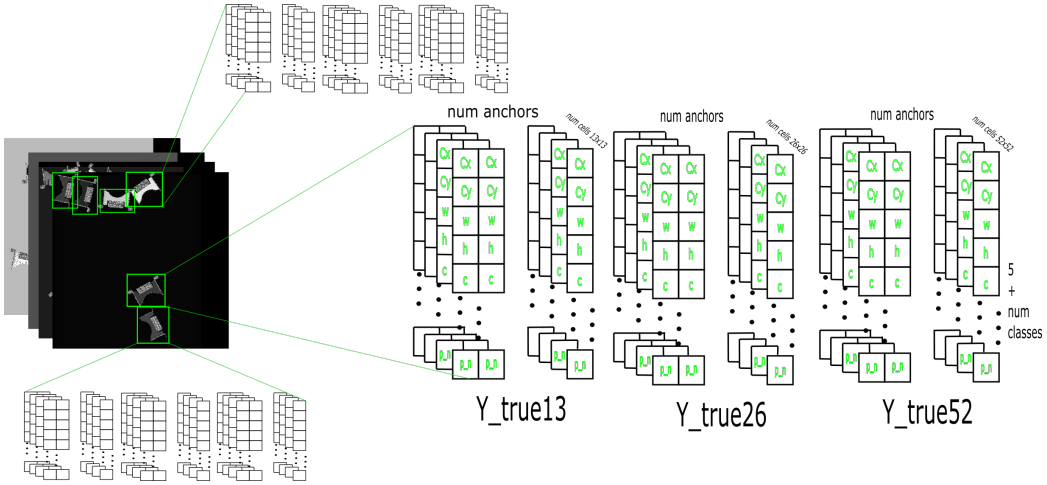


Figure 4.3: From labels boxes (ground truths) to coordinates usable by the model

4.1.3 Feature maps, predictions and loss

The YOLO forward layer provides three feature maps:

- feature map 1: image grid size equal to 13×13, it means that the image is divided into 13×13 cells, each one encoding the local features;
- feature map 2: image grid size equal to 26×26;
- feature map 3: image grid size equal to 52×52.

The loss function is computed for each feature map extracted:

- Loss 1: use feature map 1 as prediction and $y_{true}13$ as ground truth;
- Loss 2: use feature map 2 as prediction and $y_{true}26$ as ground truth;
- Loss 3: use feature map 3 as prediction and $y_{true}52$ as ground truth;

Before computing each loss, each feature map is input to the re-organization layer in order to split the bounding boxes parts. The ground truth are provided already in such form.

The final loss is $Loss = Loss1 + Loss2 + Loss3$; such value quantifies how far are the prediction from the ground truth, the goal is to minimize it.
Finally, the feature maps feed the prediction layer in order to obtain the final predicted bounding boxes.

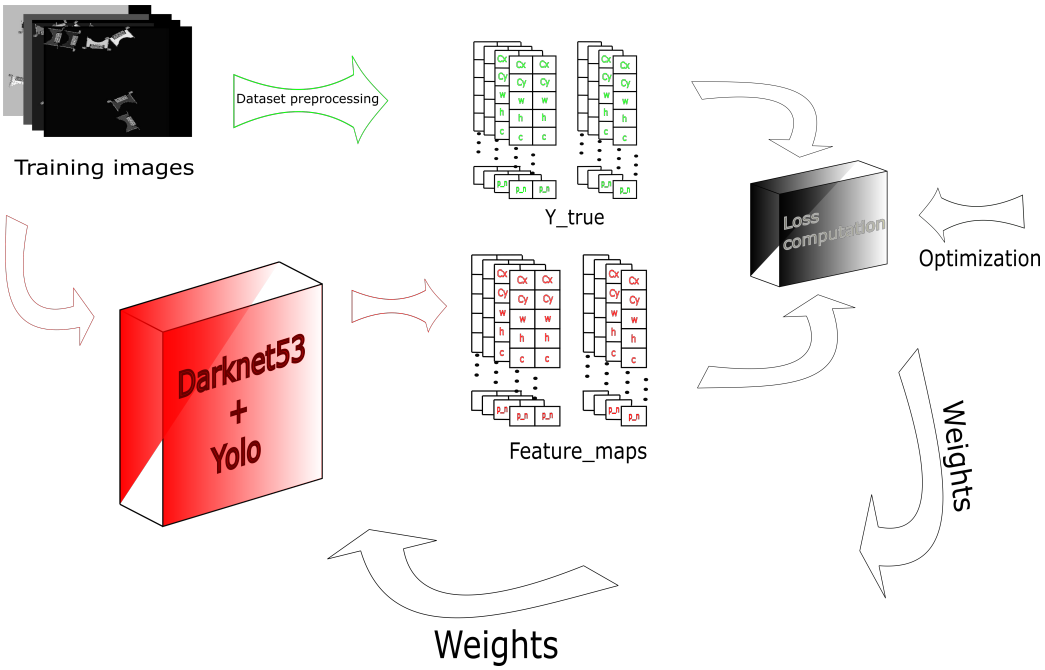


Figure 4.4: Training process

4.1.4 Optimization

The loss, once computed, undergoes a process of optimization, namely, at every loop of training, the model parameters are updated in order to decrease the loss value. Crucial for this operation are the optimizer and the learning rate used. In this case, Adam optimizer was used exploiting an exponential decayed learning rate:

$$\alpha(i) = \alpha_0 \lambda^{\frac{i}{n_{steps}}}$$

where:

- $\alpha(i)$ denotes the learning rate at step i;
- α_0 denotes the stating learning rate;
- λ denotes the decay rate;

- n_{steps} denotes the number of decay steps.

The learning rate is not constant, but gradually (exponentially) decreases; the more is the approach to the loss minimum, the less is the learning rate decreasing; the learning rate (which indicates how much the gradient has to decrease along the loss function) is correlated to the distance from the minimum value loss.

4.2 Validation

The following process is the validation; the training provides the model parameters (weights) that allow the model error to be minimum. But the training is not enough, because the parameters learned could be relative to too specific features due to the fact the training files could not generalize enough the target data. In practice, the model could undergo the 'overfitting' phenomena.
The validation process consists on feeding the model with new fresh data, obviously of the same type of the training ones, without decreasing the loss, using the weights learned during the training. For such data, the loss is computed: if the loss is of the same quantity order than the final training one, the model best learned the object features; otherwise, if the validation loss is too high, the training should be repeated with new data (if the training files are bad) or continued with more data (if the training files are good but not enough).
Such operation is performed after each training epoch; in this way it is possible to monitor, at each step, the training loss and the validation loss in such a way to easily figure out the loss decreasing and an eventual overfitting.

4.2.1 Non-maximum suppression

When the model predict the bounding boxes, it usually happens that, around the object to detect, more than one box is predicted. This is due to the fact that, in the loss function, the confidence score tends to be equal to the intersection over union of the prediction with the ground truth, if it is greater than a predefined threshold (0.5). In proximity of the object, there are more than one cell in which is predicted more than one box with great IoU. As a result, there are duplicates.
To remove such duplicates, an operation called "non-maximum suppression is performed". It works in the following way:

1. make a descending sort of the boxes based on its confidence score;

2. consider the first box (i.e. the one with highest confidence score);

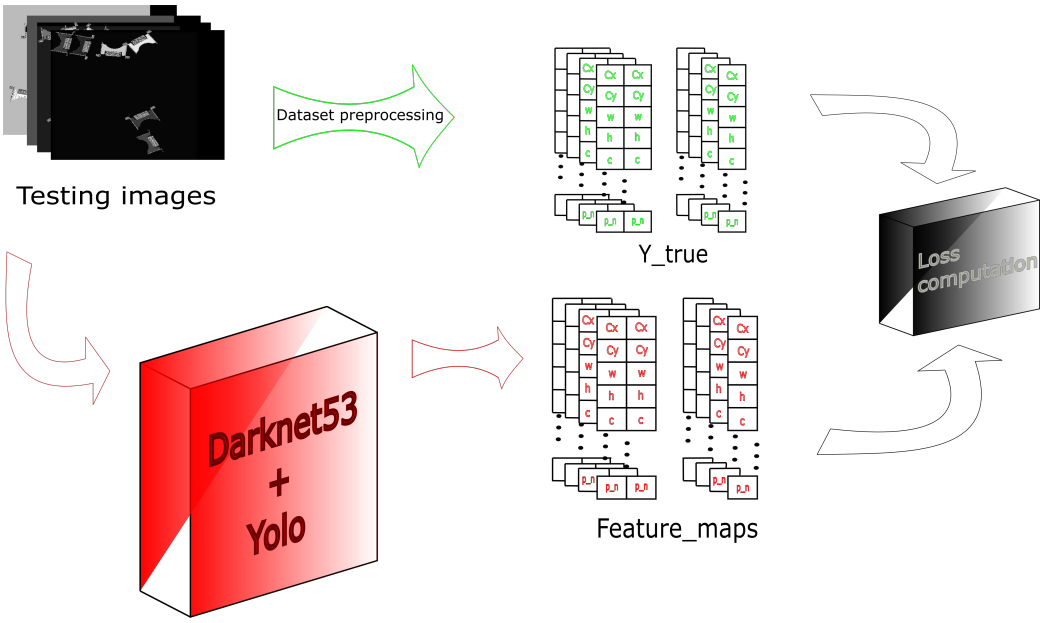


Figure 4.5: Validation process

3. compute the IoU with all other boxes;

4. discard all the boxes with IoU greater than a predefined threshold (named IoU threshold);

5. update the boxes array eliminating the discarded box (and the one considered);

6. repeat the operation with new boxes list. Stop when the update list is empty.

4.3 Info about OS, Tensorflow and GPU

Let's provide some informations about the software environment exploited to train and test the model. All the steps are written in python, through the editor Pycharm; Tensorflow library is exploited to train the neural networks, Numpy, Pillow and others are used for the preprocessing. The operative system used are both Linux Ubuntu and Windows 10.

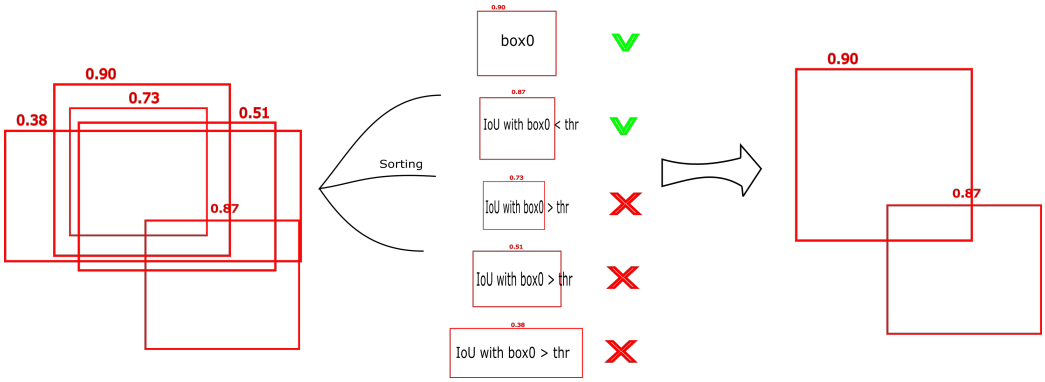


Figure 4.6: Non-maximum suppression, threshold (thr) set to 0.4

4.3.1 Programming language

All the algorithm was written in Python 3.5, in Pycharm editor. Several libraries were exploited, the most important are:

- Numpy for variables, arrays and matrix definitions with relative operations;
- Tensorflow for all that concerns with neural networks model;
- Pillow for image processing;
- Open3d for point clouds processing.

4.3.2 Tensorflow and GPU

All the deep learning model is written exploiting Tensorflow-GPU 1.11.0 with CUDA 9.0 and Cudnn 7. Due to faster computations reasons, all training and testing were performed exploiting the graphic board (GPU) instead of the CPU. Tensorflow is an open source powerful software library developed by Google, which allows to easily code and train a deep learning model, thanks to its functions like tf.slim or tf.nn; the base working consists on saving some variables (in this case, the training images) in apposite structures named Tensors. Each tensor follows a precise configuration: [batch size, image width, image height, image depth], where:

- batch size: denotes the number of images; indeed, Tensorflow allows to process groups (batches) of images at the same time;
- image width and image height: denote what the variable names say;

- image depth: stores the image channels (RGB).

Tensorflow provides several operations similar to numpy ones, that allow to operate with tensor variables.
All the Tensorflow operations are performed by the Graphic Processing Unit, thanks to Cudnn and Cuda library (developed by Nvidia), which allow to translate the high level code onto GPU language.
Two GPUs were exploited during the training and testing of the model: Nvidia GeForce 720M and Nvidia Titan V, with non-irrelevant differences on the computation speed.

4.4 Results: confidence, loss, accuracy

A detailed description of the model was provided so far; at this point, we are going to explain the concrete results, in terms of confidence, loss, accuracy, computation time, obtained by exploiting it.
First of all, it's needed to identify which are the training files, the testing files and the real files, and the not negligible differences between each other:

- training files: the images used to train the model; their creation is explained in the previous sections;

- testing files: the images used to test the model once the weights generated by the training process are available;

- real files: the images obtained from the original detection; the training files must represent as faithful as possible such data.

4.4.1 Training process

4.4.2 Confidence

Training

The confidence score indicates how much is the probability that the predicted bounding box contains a target object. During the training process, i.e. the loss minimization, the predicted confidence score tends to be equal to the intersection over union of the prediction with the ground truth. The confidence loss minimization follows a parabolic decreasing trend, with some peaks during the first steps.

Testing

During the testing, there's not loss minimization, the testing examples feed the model parametrized with the weights learned until that moment. Its trend faithfully follows the training descending parabolic trend, except in the initial steps, of course due to enough information missing. We can notice this from the initial overfitting occuring during the first 80 steps.
The loss trend does not present overfitting after the convergence; we can realize it from the plot and from the testing images.

Real

The real images are the images obtained from the detected point cloud; the goal of the training images is to simulate, as faithful as possible, the object we want YOLO to detect. The results obtained are pretty good: the target objects are all detected, some of them also with very good precision and confidence score. The results of the detection is shown in the images. In order to isolate only the right boxes, it has be set a confidence threshold and an IoU threshold; such values need to be goodly chosen since the objects could be overlapped. Indeed, the bounding box of the overlapped objects could be recognized as duplicated during the non-maximum suppression; the images show three cases:

- 0.05 confidence score threshold and 0.7 IoU threshold;
- 0.3 confidence score threshold and 0.3 IoU threshold;
- all boxes.

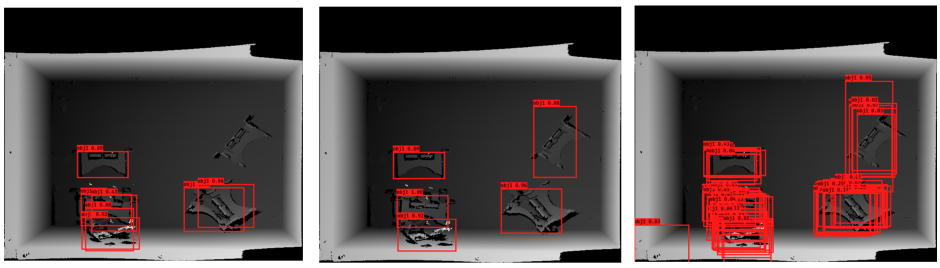


Figure 4.7: Detections

4.4.3 Loss

The loss function trend is shown splitted in its four terms: confidence, class, coordinates and sizes loss. Looking at the plots, we can make some comments; let's go to consider each term.

Confidence loss Such trend is already explained in previous paragraph; however, it's possible to explain it more in detail.

- Steps $0-100$: the initial part of the training denotes, obviously, very large values, which indicate that the predicted confidence score is still too far from the real one.

- Steps $100-500$: in this part of the training, the distance between ground truth and prediction gradually decreases, but the values are again too high. It is also possible to denote an isolated local peak of the loss function. The decreasing speed depends on the learning rate.

- Steps $500-$: the last steps of the training denote a further decreasing of the loss function. The predicted confidence becomes ever closer to the real one and the model learning becomes more and more precise. In the end, the loss function reaches the final convergence; here the loss value is small enough to conclude that the function is minimized. However, we still cannot decide if the model has well learned the image features; we need to have a look to the testing loss.

The testing loss trend is almost similar to the training one. Let's analyze each part.

- Steps $0-100$: in the first part the testing loss is very different than the training one. This is obvious: while the training immediately starts to minimize the loss, the testing undergoes an initial divergence, due to the fact that the learned parameters do not contain enough information to perform good detection. In technical terms, there occurs an initial overfitting phenomena.

- Steps $100-500$: the learning proceeds and the model starts to accumulate more and more information. This is notable by looking at the plot, which shows the testing loss stopping overfitting and starting to gradually approach the training one.

- Steps $500-$: The last part shows the convergence of the testing loss. It becomes more and more closer to the training loss; finally, it reaches the convergence to a small enough value. The final convergence deviation from the training one can be considered negligible (no overfitting). In this way, it is possible to conclude that the model has rightly learned the images features to perform good detection.

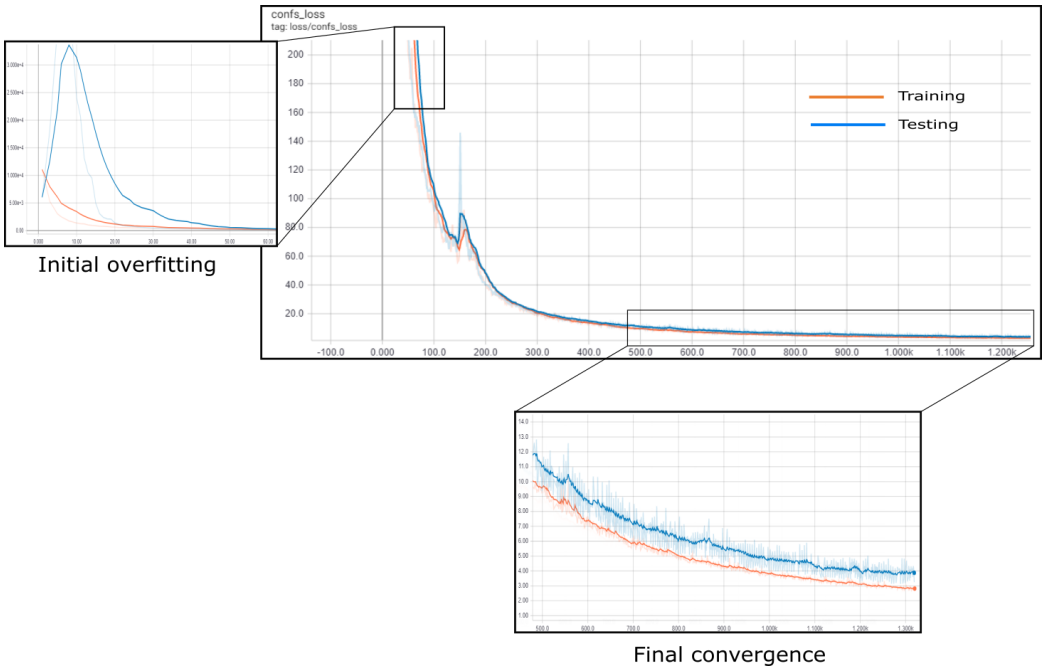


Figure 4.8: Confidence loss

Class loss Since there is only one class to predict, the class loss trend is quite simple. As in confidence case, the loss starts with high values and then gradually decreases until it reaches the convergence approximately to zero (order of 10^{-3}. The descend is parabolic without local peaks or particular deviations.
The class testing loss presents an initial overfitting in the first 100 steps; it the following steps it returns to retrace the training loss, until the convergence to zero. Also in this case, no overfitting occurs at the convergence.

Coordinates loss The coordinates loss denotes the trend of the bounding boxes centers coordinates x and y. Let's analyze it.

- Steps $0-100$: the loss values are high and start to gradually decrease, namely the predicted boxes become ever closer to the ground truths. The boxes that are modified are only the ones inside which there is an object to detect (IoU > threshold).
- Steps $100-500$: the loss decreases without particular peaks or deviations. Compared to other trends, here the descent is more linear.
- Steps $500-$ the losses converges and reaches small values.

Let's now analyze the testing loss.

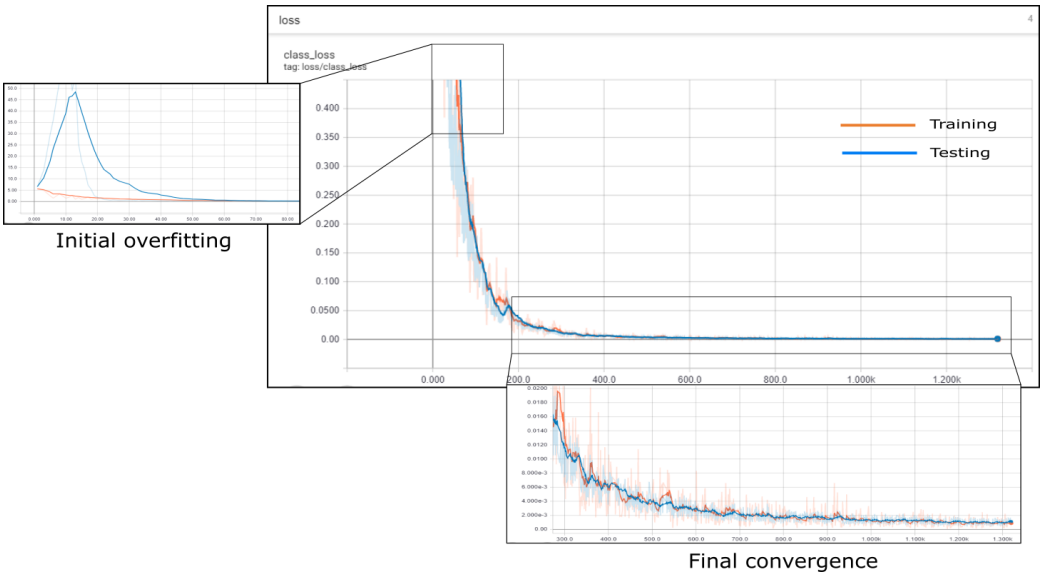


Figure 4.9: Class loss

- Steps 0 – 100: there is an initial overfitting due to not enough information for the learning.
- Steps 100 – 500: the loss stops overfitting and retraces the training trend.
- Steps 500–: the loss converges and minimizes. However, in this case, there occurs a light overfitting, even though not so relevant; indeed, the deviation from the training is greater compared to other loss trends, but it's still small and not great enough to consider a modification of the training data.

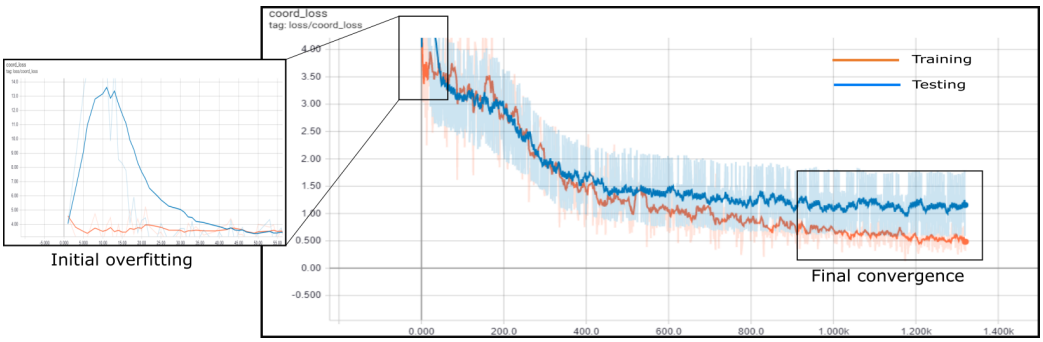


Figure 4.10: Coordinates loss

Sizes loss The sizes loss denotes the trend of the bounding boxes width and height w and h. Let's give a look in detail.

- Steps $0-100$: also in this case, there is an initial overfitting due to not enough information for the learning.
- Steps $100-500$: the overfitting stops, but occurs a huge local peak in both training and testing losses.
- Steps $500-$: loss function comes back from the local peak, minimizes and converges; no convergence overfitting occurs.

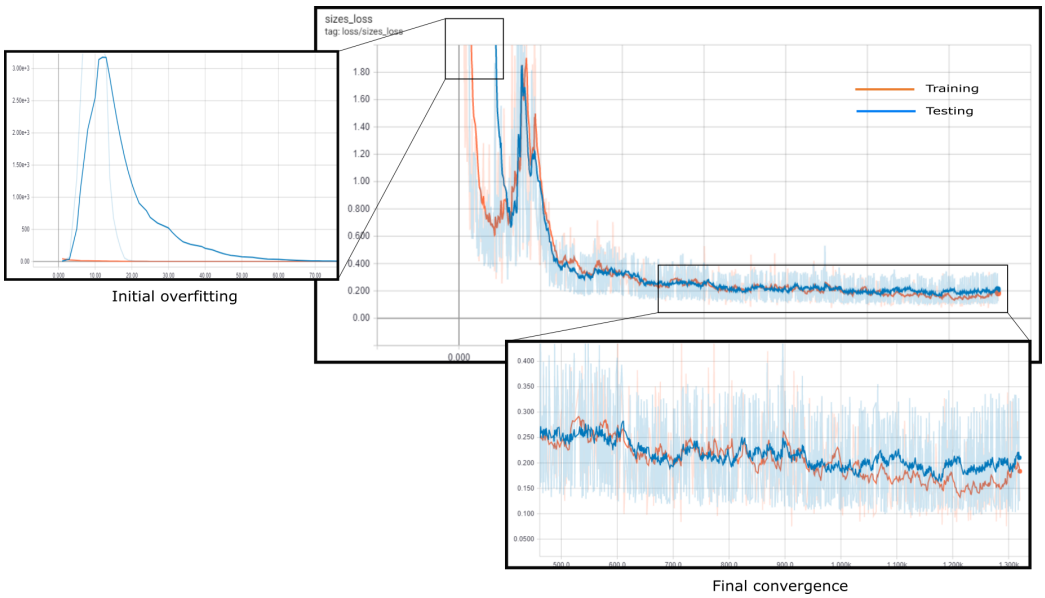


Figure 4.11: Sizes loss

4.4.4 Accuracy

Testing

Let's give a look to the detection on the testing files. The target objects are predicted very carefully, all the bounding boxes have a prediction confidence equal to 1.00 and a almost perfect overlapping with the ground truths. We can conclude that the detection on the testing image is very satisfactory.
Small clarification: some overlapping objects in the images could be undetected; this occurs because the bounding boxes are recognized as duplicates according to the non-maximum suppression threshold. So, the model is able

to recognize the overlapping objects, but it's necessary to find a compromise to set the thresholds in order to do not have too many detection in the image.

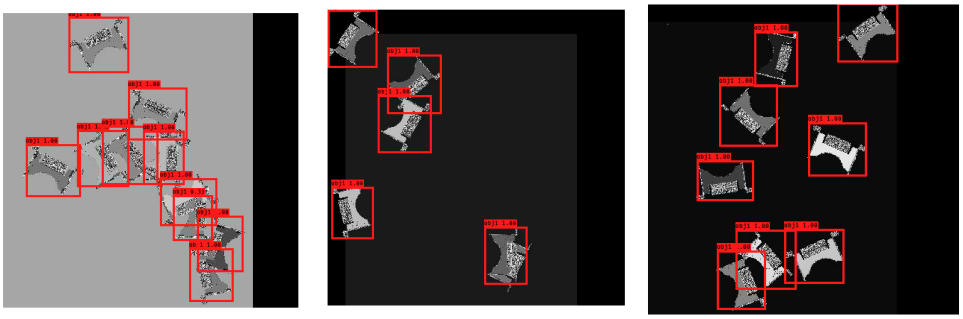


Figure 4.12: Detection on testing images

Real

The real images show a bit worse prediction, if we compare them to the testing one. Since such images are a bit noisier, the features are less precise compared to the testing. However, the objects are all well detected, the precision is a bit worse and the confidences are lower. It's possible to conclude that also in the real images the detection is satisfactory.

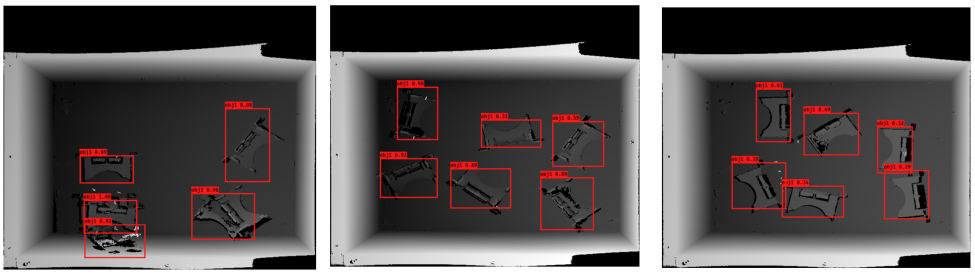


Figure 4.13: Detection on real images

4.4.5 Computation time

Let's give a look to the computation time. The model was trained and tested with tensorflow-gpu 1.12.0. To exploit the GPU, it is needed CUDA software; the one utilized is version 9.0. The learning was performed in two different GPUs: Nvidia Geforce 720M and Nvidia Titan V.

Nvidia Geforce 720M

The first exploited GPU is the Nvidia Geforce 720M. It has a memory capacity of 2048 MB (2 GB), 192 CUDA cores, 64-bit memory interface. This is a Graphic Unit without very powerful technical specifics; it's possible to find it in a common laptop.
With such GPU, all the training process (about 1300 steps) lasts about 10 hours. One step (feed the model with a tensor containing $batch\_size = 8$ images) lasts about 20 seconds.

Nvidia Titan V

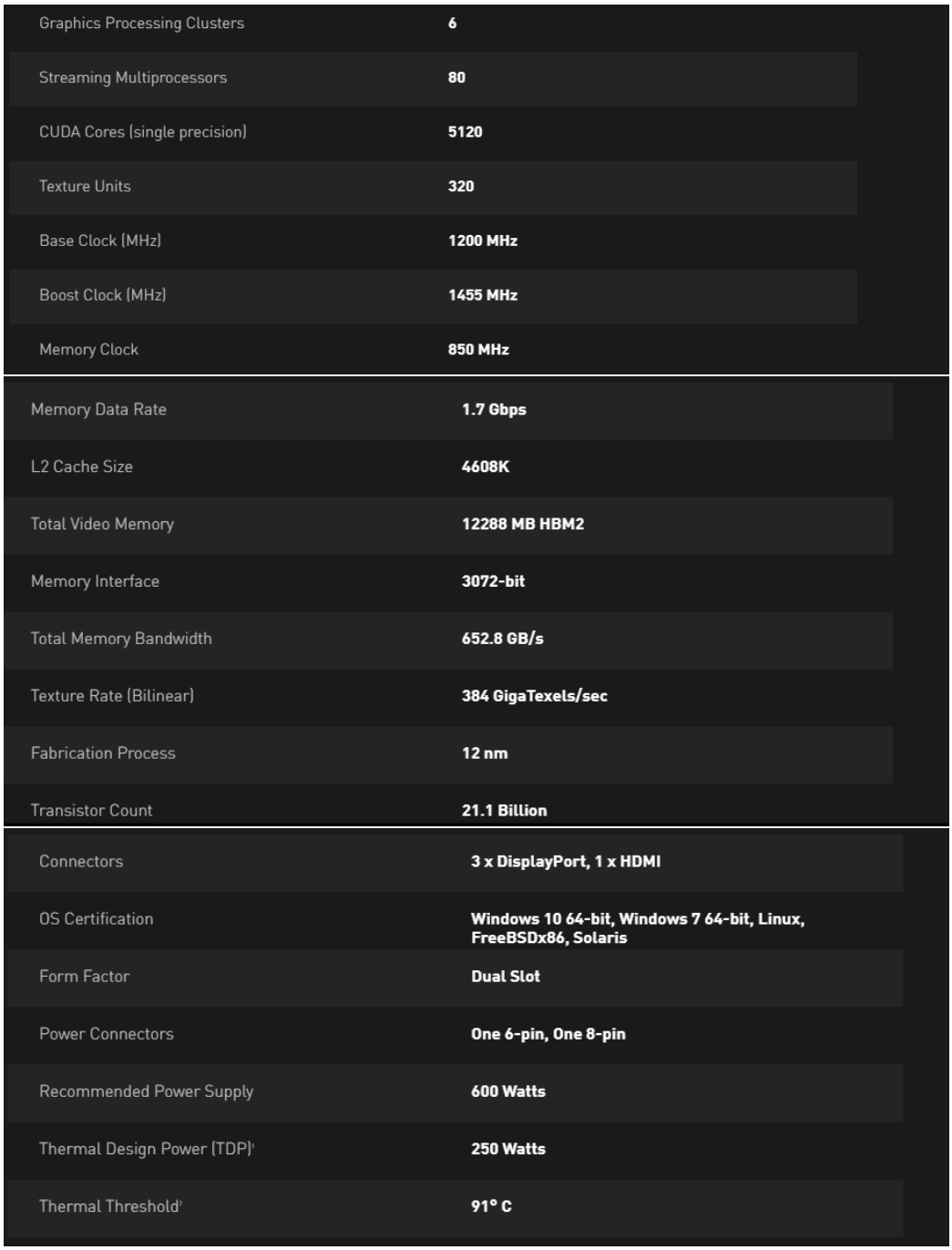
The second exploited GPU is the Nvidia Titan V. It has a memory capacity of 12288 MB (12 GB), 5120 CUDA cores, 3072-bit memory interface. This is a very powerful Graphic Unit, it's possible to find it only in expensive professional machines.
With such GPU, the training process lasts abut 1 hour and 15 minutes, with 3.5 seconds per step.

Specifications

Note: The below specifications represent this GPU as incorporated into NVIDIA's reference graphics card design. Clock specifications apply while gaming with medium to full GPU utilization. Graphics card specifications may vary by Add-in-card manufacturer. Please refer to the Add-in-card manufacturers' website for actual shipping specifications.

GT 720 GPU Engine Specs:	
CUDA Cores	192
Base Clock (MHz)	797
GT 720 Memory Specs:	
Memory Clock	1.8 Gbps or 5.0 Gbps
Standard Memory Config	1024 MB or 2048 MB
Memory Interface	DDR3 / GDDR5
Memory Interface Width	64-bit
Memory Bandwidth (GB/sec)	14.4 (DDR3) or 40 (GDDR5)
GT 720 Feature Support:	
Microsoft DirectX	12 API
OpenGL	4.4
Bus Support	PCI Express 2.0
Certified for Windows 7, Windows 8, Windows Vista, XP	Yes
Supported Technologies	CUDA, DirectX 12, PhysX, FXAA, Adaptive VSync, 3D Vision
3D Vision Ready	Yes
GT 720 Display Support:	
Multi Monitor	3 displays
Maximum Digital Resolution	3840x2160*
Maximum VGA Resolution	2048x1536
HDCP	Yes
HDMI	Yes
Standard Display Connectors	Dual Link DVI-D HDMI VGA
Audio Input for HDMI	Internal
GT 720 Graphics Card Dimensions:	
Length	5.7 inches
Height	2.713 inches
Width	Dual-width
Thermal and Power Specs:	
Maximum GPU Temperature (in C)	98
Graphics Card Power (W)	19
Minimum System Power Requirement (W)	300
3D Vision Ready:	
3D Gaming	Yes
3D Blu-Ray	Yes
3D Photos and Videos	Yes

Figure 4.14: Nvidia Geforce 720M



Graphics Processing Clusters	6
Streaming Multiprocessors	80
CUDA Cores (single precision)	5120
Texture Units	320
Base Clock (MHz)	1200 MHz
Boost Clock (MHz)	1455 MHz
Memory Clock	850 MHz
Memory Data Rate	1.7 Gbps
L2 Cache Size	4608K
Total Video Memory	12288 MB HBM2
Memory Interface	3072-bit
Total Memory Bandwidth	652.8 GB/s
Texture Rate (Bilinear)	384 GigaTexels/sec
Fabrication Process	12 nm
Transistor Count	21.1 Billion
Connectors	3 x DisplayPort, 1 x HDMI
OS Certification	Windows 10 64-bit, Windows 7 64-bit, Linux, FreeBSDx86, Solaris
Form Factor	Dual Slot
Power Connectors	One 6-pin, One 8-pin
Recommended Power Supply	600 Watts
Thermal Design Power (TDP)[1]	250 Watts
Thermal Threshold[2]	91° C

Figure 4.15: Nvidia Titan V

Chapter 5

Conclusions

Summarizing, the goal of the paper was to find a model which is able to localize some target objects inside an environment (e.g. a box). To achieve such result, two different techniques were tried: the first based on machine learning and the second based on deep learning.

Machine learning technique This method consists on the following macro-steps:

1. Feature extraction: the idea is to extract some characteristic quantity from each point (or local region) of the point cloud. In practice, for each point is calculated the covariance matrix of its $k-$neighborhood (with $k = 4$ or $k = 8$). From such matrix, the particular value M = determinant - trace is extracted. Such value allows to discriminate when a local region is a corner, an edge or a flat area, exploiting the local point variance in a smart way.

2. Classification: at this point, there is a value M for each point, hence a vector of Ms with the same cardinality of the point cloud. Such vector feeds a classifier which outputs the one probability score for each class considered; if there are 3 classes, the classifier will output 3 scores. The classifier consists on a simple fully connected neural network with one or two hidden layers (the number of layers depends on the complexity of the cloud).

3. Localization: at this point, it's possible to localize some sparse objects inside a box. The idea is to slide a box along the point cloud and compute the feature-extraction + classification for each region analyzed. The box, of course, has a predefined size, which is of the same order of the target objects size.

This technique works, but it's not advisable to perform such model if the point cloud has a large number of points, as it mainly occurs. In our case, the point cloud we have to analyze is of the order of million points. The computation time would be too high, need to find another system; let's exploit deep learning.

Deep learning technique Deep learning provides very powerful tools for object detection; such tools consists on the convolutional neural networks. Furthermore, if we join to this tools an opportune library (Tensorflow) and a good Graphic Unit, the problem becomes more manageable. So, the steps are the following.

1. Point cloud preprocessing; for deep learning it's advisable to manage images instead of an array of sparse points. Hence, it's needed to perform a cloud-to-image conversion, through a voxelization and max heights extraction. The result is the corresponding z-image of the point cloud.

2. YOLO-model; consists on a cascade of convolutional neural networks, which have the task to extract local features; the more the cascade is deep the better is the features precision. The output is a tensor containing the parameters and the confidence scores of all the boxes predicted for each cell of the image. Basing on the confidence values, only the boxes bounding a target object are selected.

3. Post-processing: to remove boxes duplicates, a non-maximum suppression is performed.

This deep learning technique is still complex, but the tools it provides (Tensorflow library) and the exploiting of a good GPU allow to well-manage the problem with good accuracy, confidence scores and, in particular, computation time.

In conclusion, the best approach to solve a common detection problem consists on deep learning techniques. Surely, this work is not perfect and could not be the most efficient way. Some useful modifications to this paper could be

- the improvement of the computation time, for example writing the model in *C* instead of python;

- modification of the detection and re-organization layers in order to make the model able to detect the objects orientation (e.g. exploiting the sine and the cosine of the angle).

Bibliography

[1] Euclid Labs, *https://www.euclidlabs.it/*

[2] Main Yolo structure, *https://github.com/YunYang1994/tensorflow-yolov3*

[3] Backpropagation, GD and SGD optimization: Computer Vision slides (Ghidoni-Carraro, 2018), Machine learning slides (Vandin, 2017)

[4] Adam optimization, *https://engmrk.com/adam-optimization-algorithm/*

[5] Momentum optimization, *https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/*

[6] Classification and regression *https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/*

[7] Yolo background, *https://pjreddie.com/darknet/yolo/*

[8] Artificial Neural Networks, *https://en.wikipedia.org/wiki/Artificial_neural_network*
Artificial Neural Networks, *http://neuralnetworksanddeeplearning.com/chap1.html*
Artificial Neural Networks, *https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/*

[9] Fully Connected Neural Network and activation functions: Computer Vision slides (Ghidoni-Carraro, 2018)

[10] Convolutional Neural Network, feature maps, padding, pooling: Computer Vision slides (Ghidoni-Carraro, 2018)

[11] Convolutional Neural Network, *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*

[12] Machine learning and deep learning image, *https://www.sumologic.com/blog/machine-learning-deep-learning/*