MASTER THESIS

Implementation and Optimization of You-Only-Look-Once (YOLO) architecture for forestland fire detection in accelerator-based platforms

Σχεδίαση και βελτιστοποίηση συστήματος You-Only-Look-Once (YOLO) για την ανίχνευση πυρκαγιών βασισμένο στην επιτάχυνση του υλικού

Supervisor:

Prof. Bellas Nikolaos

nbellas@e-ce.uth.gr

2nd Committee Member:

Assoc. Prof. Katsaros Dimitrios

dkatsar@e-ce.uth.gr

3rd Committee Member:

Prof. Lalis Spyros

lalis@e-ce.uth.gr

Student:

Christos Vasileiou

chrivasileiou@uth.gr

A thesis submitted in fulfillment of the requirements for the degree of Diploma Thesis in the Department of Electrical and Computer Engineering
at University of Thessaly
Volos, Greece

Implementation and Optimization of You-Only-Look-Once (YOLO) architecture for forestland fire detection in accelerator-based platforms

Σχεδίαση και βελτιστοποίηση συστήματος You-Only-Look-Once (YOLO) για την ανίχνευση πυρκαγιών βασισμένο στην επιτάχυνση του υλικού

Supervisor:

Prof. Bellas Nikolaos

nbellas@e-ce.uth.gr

Student:

Christos Vasileiou

chrivasileiou@uth.gr

2nd Committee Member:

Assoc. Prof. Katsaros Dimitrios

dkatsar@e-ce.uth.gr

3rd Committee Member:

Prof. Lalis Spyros

lalis@e-ce.uth.gr

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την …………….

………………..	………………..	………………..
Ν. Μπέλλας	Δ. Κατσαρός	Σ. Λάλης
Καθηγητής	Αναπληρωτής Καθηγητής	Καθηγητής

Dedicated to

my family

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Σχεδίαση και βελτιστοποίηση συστήματος You-Only-Look-Once (YOLO) για την ανίχνευση πυρκαγιών βασισμένο στην επιτάχυνση του υλικού

Περίληψη

Το ζήτημα των πυρκαγιών είναι ένα μείζων και διαχρονικό πρόβλημα που απασχολεί μεγάλο μέρος της κοινωνίας, έχοντας αρνητικές συνέπειες τόσο για το περιβάλλον (απώλειες ζώων, φυτών, κτίσματα και αντικείμενα πολιτιστικής κληρονομιάς) όσο και για την ίδια την κοινωνία, καθώς δεν είναι λίγες οι στιγμές, όπου οι απώλειες ανθρώπινης ζωής έχουν στιγματίσει τέτοια γεγονότα. Πολλές προσπάθειες έχουν γίνει για την ανάπτυξη τέτοιων εφαρμογών ικανές να περιορίσουν την αποτελεσματικότητα τους, ανιχνεύοντας τες εκ προοιμίου. Εφαρμογές όπως, η ανίχνευση από δορυφόρους περιορίζει την ελαστικότητα και την συχνότητα της αξιολόγησης της φωτιάς, μη παρέχοντας συχνή ενημέρωση με λεπτομέρειες. Η λύση σε αυτό το πρόβλημα είναι, η εφαρμογή κυκλώματος ανίχνευσης πυρκαγιών σε πρώιμο και όψιμο στάδιο στο επίπεδο της επιφάνειας της Γης, μέσω οπτικών ερεθισμάτων εκμεταλλευόμενο τον συνδυασμό δύο επιστήμων. Των νευρωνικών δικτύων και συγκεκριμένα την οπτική υπολογιστών, και την επιτάχυνση των υπολογισμών της μέσω του υλικού. Η παρούσα μέθοδος βασίζεται στην αρχιτεκτονική του You-Only-Look-Once (YOLO) αλγορίθμου, η οποία είναι ένας ανιχνευτής αντικειμένων (object detector) ο οποίοι ανιχνεύει αντικείμενα τα οποία έχουν συγκεκριμένα χαρακτηριστικά όπως χρώμα, σχήμα, μορφή, κτλ. Η επιτάχυνση και ο υπολογισμός της αρχιτεκτονικής γίνεται με τη χρήση της Edge TPU και της συσκευής Coral Dev Board της Google, η οποία έχει το πλεονέκτημα να εκμεταλλεύεται την ταχύτητα μιας TPU χρησιμοποιώντας πολύ χαμηλή κατανάλωση ισχύος.

UNIVERSITY OF THESSALY

Department of Electrical and Computer Engineering

Implementation and Optimization of You-Only-Look-Once (YOLO) architecture for forestland fire detection in accelerator-based platforms

Abstract

Wildfires is a major social problem considering that the aftermath can be costly for the environment (losses of animals, properties, cultural heritage) as well as for the society, causing loss of human life. Many efforts have been made in order to develop applications or systems able to detect wildfires early and, thus, prevent or even limit their negative consequences. Using satellites may limit the frequency of wildfire detection and as a consequence the correct and timely assessment of the situation. The solution is to use a system capable of timely fire detection on the surface near the fire spot through visual stimulation, using the conjunction of two disciplines of the computer science. Artificial intelligence, (and specifically, computer vision) in conjunction with the deployment of hardware accelerators and embedded systems. My thesis is based on the architecture of the You-Only-Look-Once (YOLO) algorithm which is used to detect objects with specific features like color, shape, form, etc. The architecture acceleration runs on the Edge TPU of the Coral Dev Board device developed by Google, having the edge over other TPUs that it runs extremely power efficient.

Acknowledgements

I would like to express the deepest appreciation to my thesis supervisor Prof. Nikolaos Bellas, who was always available and willing to help whenever I ran into trouble or had a question about my research and guided me in the right direction whenever he thought I needed it.

To my friends (and fellow students) in Volos, thank you for all your support specially during the last year of our studies. We shared experiences and moments that I will never forget.

Finally, I would like to express my acknowledgements to my family who believed in me and supported me through my years in the University of Thessaly and especially at my last year since I identified with two herniated disks, causing many mobility issues.

Christos Vasileiou
Volos, 2020

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis Idea

The outcome of wildfires is a problem which agonized the society about people safety, the flora and fauna and environmental issues. There are plenty of examples in which wildfire resulted in countless losses not only of flourish and lush areas but also of people, animals, properties, cultural heritage and legacy. For example, few weeks after the well-known huge calamity on July of 2018 in Mati of Greece it was published and uploaded a video from a closed-circuit television surveillance video (CCTV) of a house very close to the spot where the fire started. The fire spread rapidly without any interference of the local fire department, especially since that area was full of flammable trees which get fire very easily and produce a great amount of heat, causing damage. The main mistake the departments asserted was, the delayed and no qualified identification. Until now, the main problem of early fire detection is the fast, analytical and reliable report to the departments which are responsible for dealing with these kind of issues.

An approach to this problem may be the use of neural networks in aspect of detecting fire, smoke or even life beings (people, animals) close to this wildfire, in order to inform and report timeliness and reliable the departments which are in charge. There are plenty of detectors are trained to detect objects which are described as well-formed shapes with specific features. My thought process started, considering that, the fire and smoke may be the “objects” with exact characteristics like, red color for fire and on the other hand, silvery-grey color and vertical shape for smoke, driving into the need of using an architecture with a great feature extractor. Moreover, speed plays a major role to reliability and the significance of when and how quickly the fire services and the safety services should move in order to prevent a disaster. These reasons lead to the fact that a fast, reliable and precise detector may be capable to ensure an early fire detection and for all these, You-Only-Look-Once (Yolo) algorithm implementation is proposed. Yolo is an algorithm that it’s comprised of convolutional neural networks for object detection, having one input but extracting multiple outputs.

Additionally, the approach. that I recommend, relies on a single-board computer is called Google Dev Board with an on-board Edge TPU coprocessor providing a completed system in conjunction with a Camera, which observe the area that is going to be protected. They may be installed to the surface of the earth, for example, to the mountain's and hill's peaks observing mountain's sides, to long and high columns in elevated points in a city, overseeing the local area or even better to drones or general Unmanned Aerial Vehicles (UAVs) could endowed with computer vision ability, through deep neural networks.

1.2 Thesis Structure

This Thesis is divided into 5 main chapters, each one of those include smaller sections and possibly subsections. Chapter 2 provides a background of the You-Only-Look-Once algorithm and implementation. It explains in detail its components, loss function, and evaluation metrics. Chapter 3 give information about Tensorflow and Tensorflow Lite frameworks, analyzing the method of quantization. Chapter 4 provides a background of Google Coral Dev Board and Edge TPU. Chapter 5 describes the workflow that was followed, analyze issues that were addressed, provides the training strategy of the network and its performance. Finally, Chapter 6 concludes the thesis, and the future work.

1.3 Related Work

Scientists have been working on many project for fire detection for many years, based on visual stimulation taken from satellites [20][21]. This has the advantage of covering a very big parcel of land. Moreover, the addition of infrared cameras has boosted the overall precision of their project, making it extremely reliable in its detection. Infrared cameras have the edge on traditional cameras by creating images using infrared radiation and thermography. But, the distance between earth and satellites create a bottleneck of the frequency of taking and giving information [20]. This leads to new generations of satellites and more expensive approaches. It only locates the detection, without supervising the direction of local areas in more detail. The thesis proposes a low-cost approach ensuring a high reliability and frequency with detailed information of the fire direction.

Chapter 2

Yolo Object Detector

2.1. Introduction to Yolo Implementation

“You Only Look Once” is an algorithm that it’s comprised of convolutional neural networks for object detection, having one input but extracting multiple outputs. Although, it is not the most accurate and precise architecture but, it is considered as one of the most fast being real-time object detection algorithm, having greatly balanced speed and precision. Generally, an object detector often uses algorithm not only does predict class labels but also detects locations of these objects as well. So, it can detect and localize multiple objects within an image extremely rapidly. The well-known YoloV3 is an improvement over previous Yolo detection networks [1][17]. It is composed of multi-scale detection output, stronger feature extractor network is called Darknet-53, different loss function and more confident in detecting tiny object. As a result, this architecture can detect many more targets from big to small. YoloV3 is considered as one of the best models in terms of real-time object detection for 2019. However, the main disadvantage is that it cannot be implemented in currently embedded systems, by reason of high capacity and number of parameters making unavailable for these kinds of systems.

You Only Look Once Tiny (YoloV3-Tiny) is, as its name suggests, a small model implementation inspired by YoloV3 and it’s considered as a great network to implement in embedded systems and portable devices resulted in an extremely power efficient and real-time neural network detector. YoloV3-Tiny is composed of a different feature extractor (is called Darknet-Tiny) than YoloV3 and differs in using max-pooling after every convolutional layer and no use of residual networks. Furthermore, YoloV3-Tiny has similar detections outputs to YoloV3 and uses the same loss function. For the purpose of this project YoloV3-Tiny has been selected in order to be implemented in the Google Coral Dev Board, considering the number of parameters that used and the capacity of them.

2.1.1. Convolution Operation

Computer vision has been designed and is based on Convolutional Layers, and consequently, it plays a vital role in how CNNs operate. It inherits all the benefits of the Convolutional Neural Networks, which are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid. By defining, a grid-structured data, the most obvious example is a 2D image. Images exhibit spatial dependencies, because adjacent spatial locations often have similar color values. Moreover, a 3D image (e.g. RGB) holds features in more detail and it's important to maintain these spatial relationships among the grid cells, because the convolution operation and the transformation to the next layer is critically dependent on these. Each grid among D (depth size in figure 2.1) layers, is also called feature map. Each Convolutional Layer is based on a filter which convolved with a 3D grid structure, also is called convolution over volume, with shape of height, width, depth ($F_s x F_s x D$) [16].

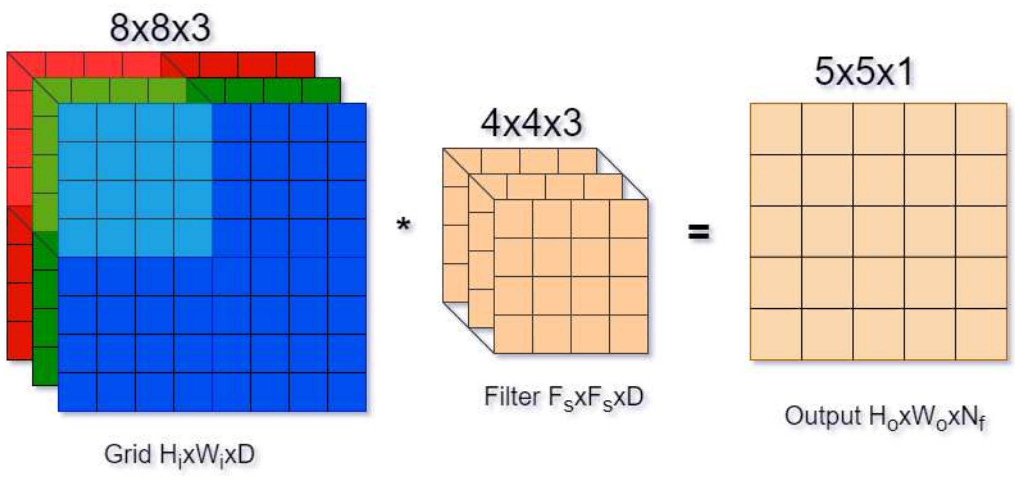


Figure 2.1: A Convolution Operation using one filter of size 4x4x3 (faded color) in a grid-structured (might be an image-RGB) of size 8x8x3.

As shown in the figure 2.1, assuming an input grid with size of $H_i x W_i x D$ and a filter with $F_s x F_s x D$, it could be also called as a kernel, a convolution operation exports an output grid of size $H_o x W_o x N_f$. Two very important notes that they are remarkable, is that:

1. a filter must have the same depth size (D) as its input and
2. output's depth is based on the number of filters (N_f) which are applied to the input.

Moreover, a filter starts to apply in the input grid, layer by layer, and each filter's cell multiplied by the grid's cell, respectively and adding each product together, resulting in a total summation of D additions. Then, the filter slides by one or more cells, which is called stride (S), and a 2nd step starts again until the filter has been recursively applied in the whole input grid. Afterward, the values are passed through an activation function like, relu or leaky relu. Frequently, there is the need to export an output with the same size as its input (H_o equal to H_i, and W_o equal to W_i). It could be possible by applying a filter with size of 1x1 or by applying the padding in the input grid. Padding (P) is the addition of extra columns (above or below) or extra rows (left or right) of zeros to the input grid [2]. The formula for the exported sizes of a convolution operation is defined as

- $H_o = \left\lfloor \frac{H_i + 2P_{row} - F_s}{S} \right\rfloor,$
- $W_o = \left\lfloor \frac{W_i + 2P_{column} - F_s}{S} \right\rfloor,$
- $N_f = number\ of\ filters$

Where, W, H are the size of input and output. P, S are padding and stride respectively. F_s the size of the filter [16].

For example, if the input is an image of size 416x416x3 (RGB-image), at the 1st Convolutional Layer, 16 filters are applied of size 3x3 (the depth size is equal to 3 as input's) with stride of 1 and two extra rows above and below, respectively to the image, and two extra columns left and right, respectively to the image then,

$$H_o = \left\lfloor \frac{416 + 2 * 2 - 3}{1} \right\rfloor = 416, \qquad W_o = \left\lfloor \frac{416 + 2 * 2 - 3}{1} \right\rfloor = 416, \qquad N_f = 16,$$

the Layer will export a volume of size 416x416x16.

2.1.2. Batch Normalization

Deep Neural Networks are complicated by the scope of training and the fact the distribution of each layer's inputs changes during training, as the parameters of the previous layer change [3][16]. This phenomenon slows down incredibly the training by requiring lower learning rates and careful parameter initialization. Batch Normalization allows to Deep Neural Networks to use much higher learning rates and be able to initialize the parameters easier, creating much more robust and deeper Networks. Moreover, it acts as a regularizer, eliminating the need for Dropout. The most important fact about Batch Normalization is that achieves a much faster training with the same or even better precision and accuracy. Batch Normalization, in practice, normalizes the parameters-weights before the activation function and not the parameters after it. According to paper of Sergey Ioffe and Christian Szegedy [3], the following formulas try to normalize and manipulate any activation function:

1. $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean
2. $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ //mini-batch variance
3. $\widehat{x_i} \leftarrow \frac{(x_i-\mu_B)}{\sqrt{(\sigma_B^2+\varepsilon)}}$ //normalize
4. $y_i \leftarrow \gamma\widehat{x_i} + \beta = BN_{\gamma,\beta}(x_i)$ //scale by γ and shift by β, where γ, β are trainable variables

The normalization of activations that depends on the mini-batch allows efficient training, but is neither necessary nor desirable during inference. But, in Convolutional Neural Networks, Batch Normalization is used before the activation function in order to make the output to depend only on the input, deterministically. For this reason, once the network has been trained, the 5th and 6th formulas make effect using the population, rather than mini-batch. Using moving averages network can track the accuracy of the model as it trains and are responsible for time inference of the network [15][16].

5. $\hat{x} = \frac{x-E[x]}{\sqrt{Var\{x\}+\varepsilon}}$ // moving_mean non-trainable
6. $Var[x] = \frac{m}{m-1}E_B[\sigma_B^2]$ // moving_variance non-trainable

These 4 variables (gamma, beta, moving_mean, moving_variance) take place in the Batch Normalization Layer and are responsible for each feature map functionality. For further explanation read [3].

2.1.3. ReLU, Leaky ReLU

The activation functions introduce the nonlinearity of the real-world properties to artificial neural networks. In Deep Neural Networks frequently are used activation functions like, the ReLU and the Leaky-ReLU. Leaky-ReLU is used in the implementation of YoloV3-Tiny network, but, ReLU is examined as it is going to replace Leaky-ReLU with the intention to convert the model (*section 4.2*). Both of them are applied to the exported volumes of the Convolutional Layers element-wisely and then passed on to the next layer [16].

ReLU (Rectified Linear Unit):

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

Leaky ReLU (Leaky Rectified Linear Unit):

$$f(x) = \max(0.1x, x)$$

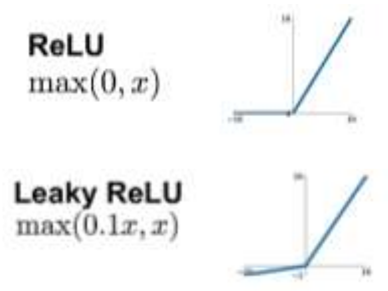


Figure 2.2: Activation's Graphs.

The advantages of ReLU are fast computation and vanishing gradient problem by meaning of squashing the propagated error signal as long as unit is activated, make possible to implement deeper networks. On the other hand, the main disadvantage is that, may waste units to converge correctly to the target, resulting in the fact that the units, which are never activated above threshold, don't learn and be saturated. This problem is solved by the use of Leaky ReLU, where slope is changed left of x=0 as shown in figure 2.2.

2.1.4. Max-Pooling

Unlike the convolution operation, max-pooling exports a volume with the same depth-size as its input, pooling only one value (max-value) in the corresponding feature map [16]. Additionally, it's usually assigned the same stride as the size of the filters in order to pool values without overlap among the different regions. However, it has sometimes been suggested that it's desirable to have at least some overlap among the spatial units, because it makes the approach less likely to overfit. Frequently, Map-Pool layer aim to gradually reduce and downscale the dimensionality of the network's volumes.

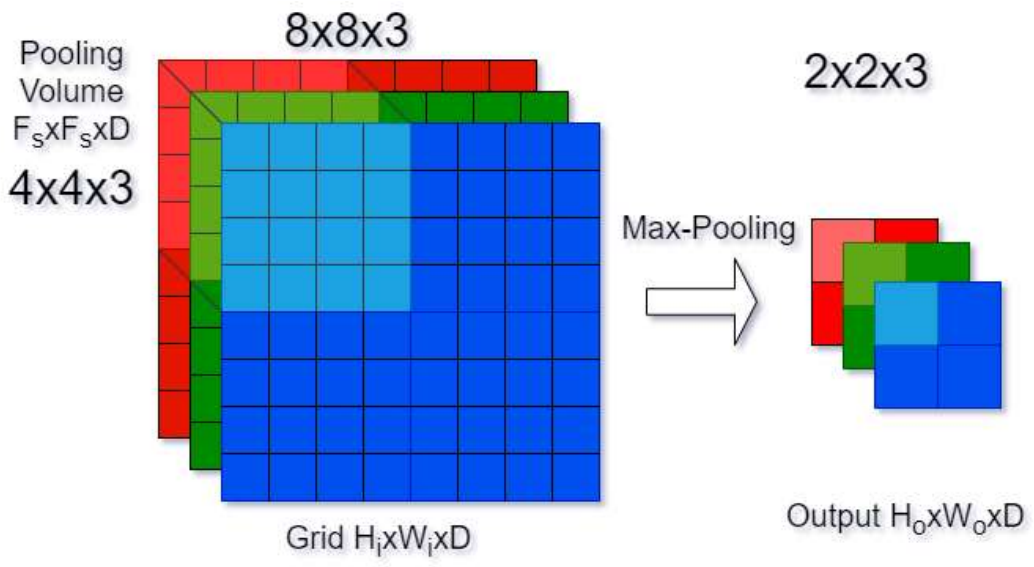


Figure 2.3: Max-Pooling operation with stride same as filter's size (filter is depicted at the faded color with size of 4).

2.2. Yolo Architecture

First of all, YoloV3-Tiny architecture can be divided into two main components:

- A backbone network is called Darknet-Tiny, and
- 2 multi-scale detectors.

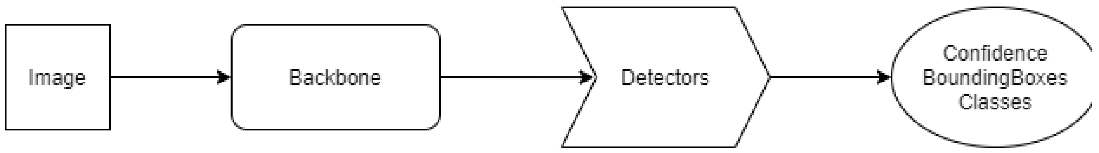


Figure 2.4: Yolo's Speculative Network Diagram.

When an image comes into the network, firstly, it goes through the backbone in order to extract the features that an image is comprised of. Secondly, detectors are activated resulting in two outputs with different scales feeding each an "extra layer" in order to encode and get the corresponding output (Confidence, BoundingBoxes, Classes). This extra layer converts each output to a logical grid in NxN (i.e. 13x13) cells and compute boxes for each cell. Afterwards, it applies the Non-Max Suppression algorithm to filter out the unnecessary boxes with lower confidence for the same object. Both of these layers will be explained in detail in section 2.4.

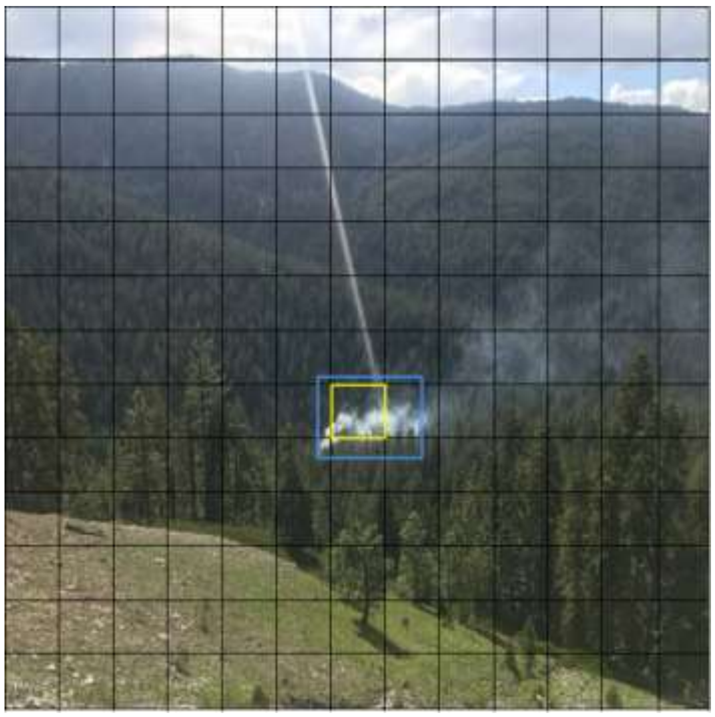


Figure 2.5: A grid of 13x13 size and the cell(yellow) responsible for a specific detection.

2.2.1. YoloV3-Tiny's backbone or Darknet-Tiny

The backbone, also known as feature extractor, that YoloV3-Tiny uses is called Darknet-Tiny. It comprised of 7 Convolutional Layers, each one is followed by Batch Normalization layer and LeakyReLU (Darknet-Tiny in figure 2.5, Conv2D_BN_Leaky in figure 2.6). Max Pooling is used after LeakyReLU causing downsampling in size of output.

Each convolutional layer uses a 3x3 filter size with stride equal to 1, as a result parameters are needed 3x3x$N_{filters}^{i-1} x N_{filters}^{i}$, where, i-1 and i are the previous and the current layer, respectively. Additionally, after convolutional layers are implemented Batch Normalization Layer responsible for the stability of the neural network and most importantly preventing the overfitting. It normalizes the weights by subtracting the batch mean and dividing by the batch

#	Layer	NFILTERS	Filter Size/Stride	Output Size	Parameters (float-32)
1	Conv2D_BN_Leaky	16	3x3/1	416x416x16	3x3x3x16+4x16 = 496
2	Max-Pool	16	2x2/2	208x208x16	-
3	Conv2D_BN_Leaky	32	3x3/1	208x208x32	3x3x16x32+4x32 = 4.736
4	Max-Pool	32	2x2/2	104x104x32	-
5	Conv2D_BN_Leaky	64	3x3/1	104x104x64	3x3x32x64+4x64 = 18.688
6	Max-Pool	64	2x2/2	52x52x64	-
7	Conv2D_BN_Leaky	128	3x3/1	52x52x128	3x3x64x128+4x128 = 74.240
8	Max-Pool	128	2x2/2	26x26x128	-
9	Conv2D_BN_Leaky	256	3x3/1	26x26x256	3x3x128x256+4x256 = 295.936
10	Max-Pool	256	2x2/2	13x13x256	-
11	Conv2D_BN_Leaky	512	3x3/1	13x13x512	3x3x256x512+4x512 = 1.181.696
12	Max-Pool	512	2x2/1	13x13x512	-
13	Conv2D_BN_Leaky	1024	3x3/1	13x13x1024	3x3x512x1024+4x1024 = 4.722.688
	SUMMARY	-	-	-	6.298.480

Figure 2.6: Analytic information of Darknet-Tiny and its parameters configuration, assuming input size 416x416. Conv2D_BN_Leaky is shown in figure 2.7.

standard deviation. Afterwards, the activation function is leaky relu constricts the negative values resulting in max pool layer.

Layer	Filters	Filter Size/Stride	Output Size	Parameters (float32)
Conv2D	$N_{filters}$	$F_W x F_H / 1$	$H_{input} x W_{input} x N_{filters}$	$F_W x F_H x N^{i-1}_{filters} x N^{i}_{filters}$
Batch Normalization	$N_{filters}$	-/-	Same as Input	$4 x N^{i}_{filters}$
Leaky ReLU	-	-	Same as input, $f(x) = \max(0.1x, x)$	-
Conv2D_BN_Leaky SUMMARY	-	-	-	$F_W x F_H x N^{i-1}_{filters} x N^{i}_{filters} + 4 x N^{i}_{filters}$

Figure 2.7: Conv2D_BN_Leaky in detail. Batch Normalization has 4 training parameters per filter as described in *section 2.1.2*

Another important advantage of Yolo is that is invariant to the input's size. The network downsamples the image by a factor called the stride of the network. For example, if the stride of the network is 32, then considering an image's size is 416x416 the architecture will output a grid of size 13x13. However, practically this might have to stick to a constant input size due to the fact that the network's head show various problems once the algorithm is implemented. For instance, when a GPU is used and it is processing an image in batch (can be processed in parallel, leading to speed boosts) it is needed to have all images of fixed size and concatenating multiple images into a large batch. Moreover,

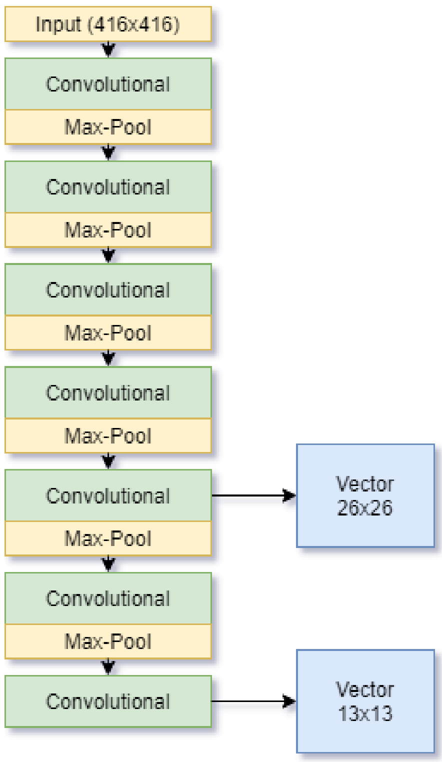


Figure 2.8: Darknet-Tiny assuming input size 416x416

the input size plays a major role for the speed of the network, because the larger an image is the more time needs the detector to apply all the filters.

2.2.2. Multi-scale Detectors

Having the outputs from feature extractor now they can be fed to detectors. There are 2 detectors each one of them contains a series of 3 Convolutional layers with the last one responsible for extracting the architecture's output. First and foremost, the first 2 Convolutional Layers (1x1x256, 3x3x512-green kernels in figure 2.9) are also called Neck of the architecture and the last one is called Head or Feature Map (white convolutional layer in figure 2.8). The first detector takes the output with the greater downsampling scale convolves it and exports a grid with a total stride of 32 (13x13 grid). The other detector convolves the output of the 1st Convolutional Layer of 1st detector with a kernel of 1x1x128, upsampling it and concatenate it with the 2nd output of the Darknet-Tiny (output from 5th Convolutional Layer). After it, there is the 3x3 Kernel and the Feature Map, resulting in an output with total stride 16 (26x26 grid). The Concatenation Layer is able to combine the features of one neck with the other, like a residual net. As the figures shown below (figure 2.9 and 2.10), the architecture's detectors provide 2 different strides of 32 and 16.

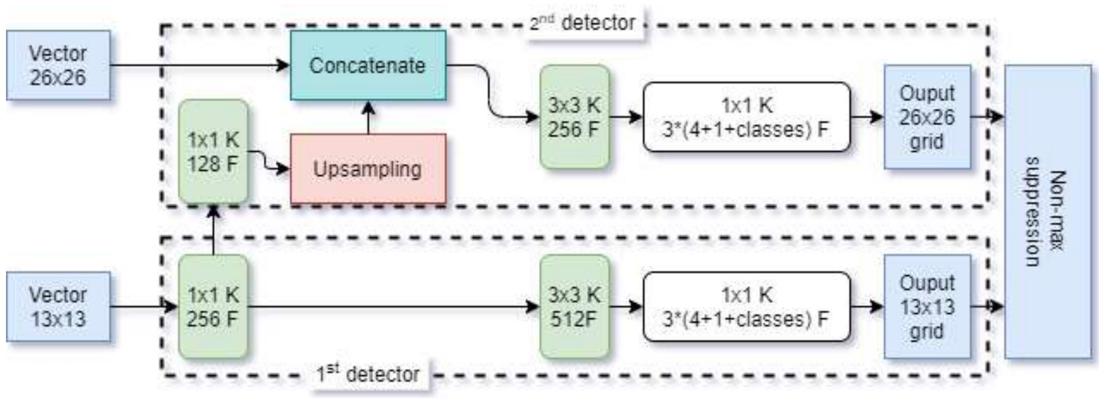


Figure 2.9: Detector's architecture with input size 416x416.
Where, K and F are Kernel and Filters' sizes respectively.

#	Layer	N_{FILTERS}	Filter Size/Stride	Input size	Output Size	Parameters (float32)
14	Conv2D_BN_Leaky	256	1x1/1	13x13x1024	13x13x256	1x1x1024x256 + 4x256 = 263.168
15	Conv2D_BN_Leaky	512	3x3/1	13x13x256	13x13x512	3x3x256x512 + 4x512 = 1.181.696
16	Conv2D	3*(4+1+cl)	1x1/1	13x13x512	13x13x3(5+cl)	1x1x512x3(5+cl) = 7.680+1536classes
	Route 14	-	-/-	-	-	-
17	Conv2D_BN_Leaky	128	1x1/1	13x13x256	13x13x128	1x1x256x128 + 4x128 = 33.280
18	Upsampling	-	-/-	13x13x128	26x26x128	-
19	Concatenate 9, 18	-	-/-	[(26x26x256), (26x26x128)]	26x26x384	-
20	Conv2D_BN_Leaky	256	3x3/1	26x26x384	26x26x256	3x3x384x256 + 4x256 = 885.760
21	Conv2D	3*(4+1+cl)	1x1/1	26x26x256	26x26x3(5+cl)	1x1x256x3(5+cl) = 3.840+768cl
	SUMMARY	-	-	-	-	2.375.424+ 2.304cl

Figure 2.10: Analytic information of Yolo Neck and Head their parameters configuration. Where classes are the number of classes that are going to be trained and detected. In project's case the classes is equal to 2 (fire, smoke) the parameters of Neck and Head are 2.380.032.

2.3. Output Interpretation

Firstly, it's needed to define two important details of architecture:

1. The input is a batch of images of shape (m, 416, 416, 3).
2. The network exports an output which is a list of bounding boxes with shape (m, grid_size, grid_size, anchors, BoundingBox)

Notes: where:

- m, is the batch's size,
- grid_size, is the each detectors size according to the strides of the network.
- anchors, is the quantity of anchors is used to detect the corresponding object. It will be explained in detail, later on.
- BoundingBox, is represented by 6 or 5+$N_{CLASSES}$ (one-hot encoding) numbers (t_{pc}, t_{bx}, t_{by}, t_{bw}, t_{bh}, t_{cl}) it will be explicitly explained, later on.

Anchor boxes have predefined aspect ratios (predefined width and height only). In order to collect the anchors boxes, the K-means algorithm is used to the entire dataset to cluster the corresponding aspect ratios. Since, the network exports the grid with bounding boxes then, the predicted values of the boxes anchor to the anchor boxes, extracting the final predictions, sharing the same centroid. As a result, creating the bounding boxes (figure 2.11). Then, it may decide which bounding box is fitted better to the ground truth box, related to the best IoU (intersection over union). This algorithm works because the anchor boxes should look like the ground truth boxes, because of clustering, and it's only needed a subtle adjustment.

YoloV3-Tiny uses 6 anchors boxes to the 2-scale output. Since, the output is multi-scaled then, they're needed to be sorted in descending order corresponding to their area and to be split into 2 triples, each for any output as shown in figure 2.12. With this concept, the tiny cells from 26x26 grid are responsible for detecting smaller objects than the cells from 13x13 grid.

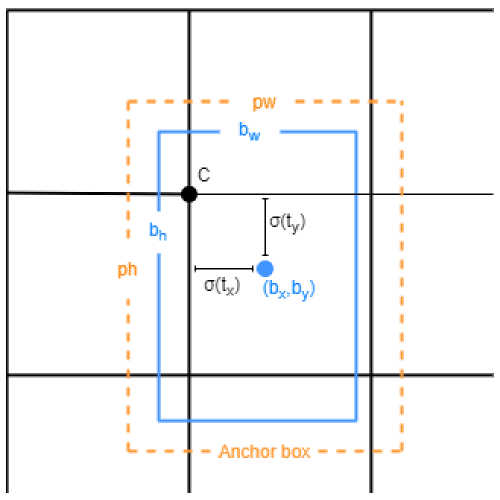


Figure 2.11: Predicted bounding boxes (b_x,b_y,b_w,b_h) is predicted using offset to ancho prior box.

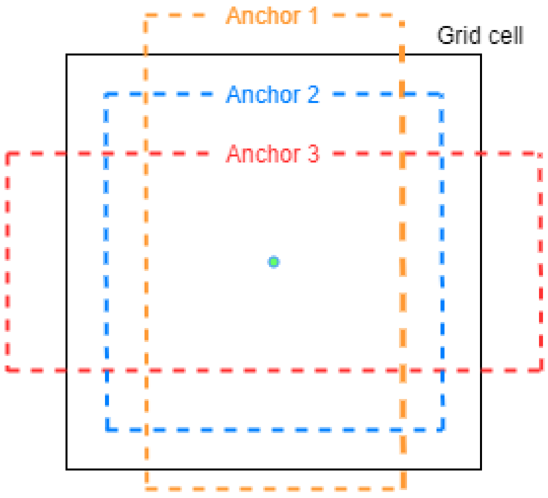


Figure 2.12: Triple of Anchor boxes in a cell.

Feature Map predicts a fixed number of bounding boxes for each cell of the grid. This fixed number is equal to the quantity of anchors are assigned to each output. In YoloV3-Tiny are assigned 3 anchor boxes per output. The consequential total predicted boxes are the addition of:

- 13x13x3 of the first output with,
- 26x26x3 of the second,
- resulting in, 2.535 bounding boxes.

The features in detectors are learned by the convolutional layers weights, which at the end, they are passed into a classifier/regressor that it forms the prediction (coordinates of the bounding boxes, the classes labels, etc.). This is the job of the Output Layer (last layer – after Feature Map). In the Output Layer, t_{pc}, t_{cl} are passed through a sigmoid function and the localization variables t_{bx}, t_{by}, t_{bw}, t_{bh} have the encoding of the figure 2.13. In each Output Layer is considered a grid of dimensions equal to that of the Feature Map. The goal is to get the bounding boxes and their classes from Output Layer. The bounding boxes contain information for localization and classification, as well. The *pc* is the class probability, defining the percentage of, whether the BoundingBox detects an object or not, and the *cl* encodes the different classes that network can detect, in logic of one-hot encoding, asserting the probability that there is a specific class. The *bx*, *by*, *bw*, *bh* are coordinates, they localize the object. The centers of the bounding boxes are encoded (bx, by) and normalized to the size of the cell, considering that, top-left corner of each cell is the point (0,0) it can also called offset from this point. The width and height (bw, bh) are encoded to scale the aspect ratios of the anchor boxes. It's worth noting that, having the predicted values, it could be possible to use Mean Square Error to compare with the ground truth boxes in order to train the localization variables, predicting them directly (without using anchor boxes). However, due to the large variance of scale and aspect ratio of boxes, it's extremely hard for the network to converge leading to unstable gradients during training. Instead, most of the modern object detectors predicts offsets to default bounding boxes, is called anchors, that are mentioned earlier.

1. $b_x = \sigma(t_x) + C_x \xrightarrow{invertion} t_x = \sigma^{-1}(b_x - C_x)$
2. $b_y = \sigma(t_y) + C_y \xrightarrow{invertion} t_y = \sigma^{-1}(b_y - C_y)$
3. $b_w = p_w e^{t_w} \xrightarrow{invertion} t_w = ln(\frac{b_w}{p_w})$
4. $b_h = p_h e^{t_h} \xrightarrow{invertion} t_h = ln(\frac{b_h}{p_h})$
5. $p_c = \sigma(t_{pc}) \xrightarrow{invertion} t_{pc} = \sigma^{-1}(p_c)$
6. $cl = \sigma(t_{cl}) \xrightarrow{invertion} t_{cl} = \sigma^{-1}(cl)$

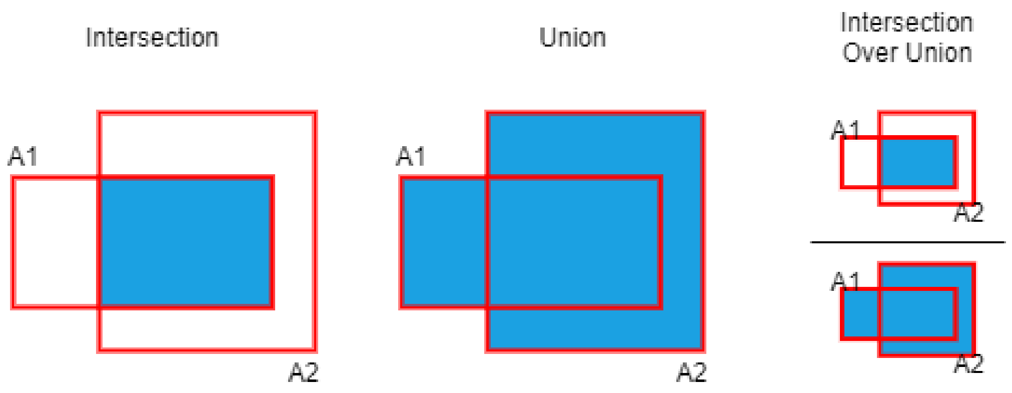
Figure 2.13: The encoding of the boxes (localization variables 1-4). Existence probability of an object (5). One-hot encoding multi-label classification probability (6).

Notes: where:

- t is the output of Head layer,
- σ(t) is the sigmoid function, and
- p is the anchor prior

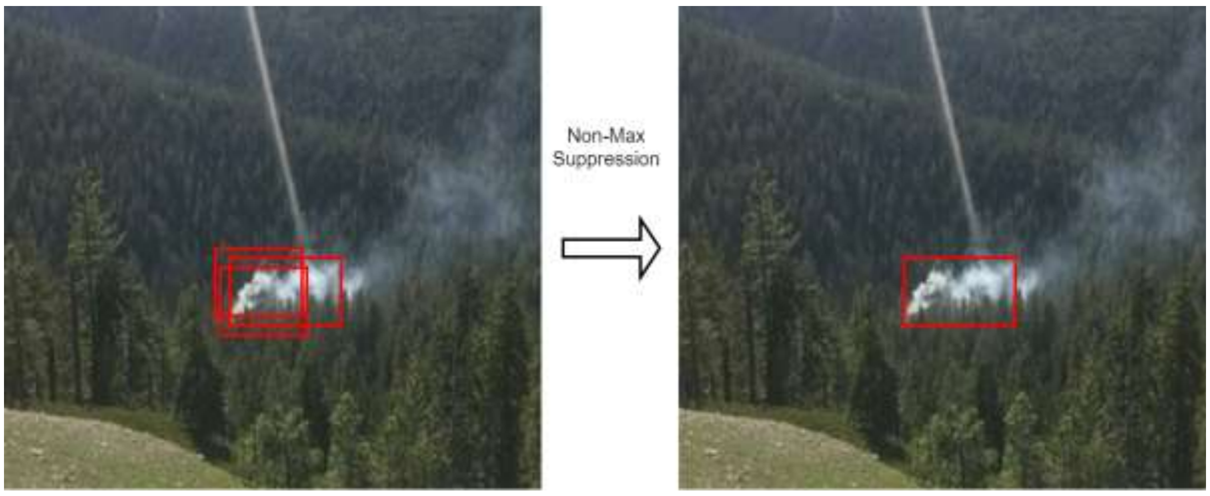
2.4. Non-Max Suppression

In case of, existing only one object in the image, the question is, how is it possible to reduce the detections from 2.535 to only 1. The answer is to apply a filter in order to keep these objects they have highest confidence score. Generally, boxes having scores below a certain threshold (for example below 0.5) are ignored. The remaining bounding boxes are more confident that they have detected some object. The next concern comes with the assertion of, what if, more than one bounding box are used to detect one object. Non-Max



$$IoU = \frac{A1 \cap A2}{A1 \ \cup A2}$$

(a)



(b)

Figure 2.14: (a) The IoU filter between two boxes is shown. (b) The Non-Max Suppresion filter is used to ignore bounding boxes with less IoU.

Suppresion intends to cure the problem by keeping only one detection for a specific ground truth box (figure 2.14b), applying an IoU (Intersection over Union – figure 2.14a) algorithm.

For instance, let's implement IoU algorithm:

Considering that, top left corner of the cell is the point (0,0) and assuming two boxes and the Intersection Box with coordinates of top-left and lower-right points, then:

1. Box 1: ($x1_{min}$, $y1_{min}$, $x1_{max}$, $y1_{max}$)
2. Box 2: ($x2_{min}$, $y2_{min}$, $x2_{max}$, $y2_{max}$)
3. Intersection Box: (xi_{min}, yi_{min}, xi_{max}, yi_{max})

$$xi_{min} = \max\,(box1[0], box2[0])$$

$$yi_{min} = \max\,(box1[1], box2[1])$$

$$xi_{max} = \max\,(box1[2], box2[2])$$

$$yi_{max} = \max\,(box1[3], box2[3])$$

$$area_{inter} = (xi_{max} - xi_{min})\,(yi_{max} - yi_{min})$$

$$area_{box1} = (box1[3] - box1[1])(box1[2] - box1[0])$$

$$area_{box2} = (box2[3] - box2[1])(box2[2] - box2[0])$$

$$IoU = \frac{area_{inter}}{(area_{box1} + area_{box2} + area_{inter})}$$

For the need of this project the score and iou thresholds have been assigned to 0.5.

2.5. Yolo Loss Function

The loss function of yolo consists of four main parts. It measures localization loss of both centroids and predicted sizes (width, height), objectness loss by measuring confidence probability and classification loss. Therefore, the following equations are implemented 3 times, each for any output.

General Notes:

- *SE()*:

  is the square error formulated by $SE(x, y) = \sum_{i=0}^{N}(x - y)^2$
- $Sparse_{categorical-crossentropy}(cl, cl')$:

  Used for multiclass classification problems, having target class encoded as an integer, and predicted classes as one-hot vector. Take the values after, they have been passed through sigmoid (cl).
- $Binary_{crossentropy}(t_{pc}, t'_{pc})$:

  Used when the network typically achieve prediction by sigmoid activation. Take the values before pass through the sigmoid (t_{pc}), which are called logits. The formula is used: $-(p(x)log(q(x)) + (1 - p(x))log(1 - q(x)))$ where, p, q are target and prediction, respectively.
- *mask_{obj}*:

  a mask which it's filtering out the cells where there are no ground truth boxes and consequently there is no need to be included in the loss. It consist of ones or zeros and this is the reason of multiplication with SE().
- *λcoord*:

  is a constant, which is giving greater values in the tiny-object. It's depended by the ground truth box's area.
- t_x, t_y, t_w, t_h:

  are the parameters of a box's encoding. As a result, there is a need the ground truth boxes to be converted by the inversions to be used with network's predictions.
- S^2:

  defines the output's size of grid.
- *anchors*:

  defines the 3 corresponding boxes of grid's cell.

- *num_classes*:

  defines the number of classes.

Losses:

1.

$$Loss_{xy} = \lambda_{coord} mask_{obj} \sum_{i=0}^{S^2} \sum_{j=0}^{Anchors} SE(t_{xij}, t'_{xij}) + SE(t_{yij}, t'_{yij})$$

2.

$$Loss_{wh} = \lambda_{coord} mask_{obj} \sum_{i=0}^{S^2} \sum_{j=0}^{anchors} SE(t_{wij}, t'_{wij}) + SE(t_{hij}, t'_{hij})$$

where:

- t'_x, t'_y are the coordinates of the centroid is predicted by network and respectively t_x, t_y are the ground truth box centroid's coordinates. The closer these centroids are to each other, the smaller the loss is.
- t'_w, t'_h are the width and height are predicted by network and t_x, t_y are the ground truth box's aspects.

3.

$$bce_{obj-loss} = \sum_{i=0}^{S^2} \sum_{j=0}^{anchors} Binary_{crossentropy}(t_{pc}^{ij}, t'^{ij}_{pc})$$

$$Loss_{obj} = mask_{obj} bce_{obj-loss} + (1 - mask_{obj}) mask_{ignore} bce_{obj-loss}$$

4.

$$bce_{class-loss} = \sum_{i=0}^{S^2} \sum_{j=0}^{anchors} \sum_{c=0}^{num\_classes} Sparse_{categorical-crossentropy}(t_{cl-c}^{ij}, t'^{ij}_{cl-c})$$

$$Loss_{class} = mask_{obj} bce_{class-loss}$$

where:

- $t_{pc}^{ij}, t'^{ij}_{pc}$, are the prediction probability of object existence and the target of object existence
- t_{cl}, t'_{cl}, are both in one-hot encoding of what class the network has predicted and the target.

During training, these 4 losses are to be minimized and make capable the network to detect the desirable objects of the corresponding dataset.

2.6. Yolo Metrics

Yolo algorithm is considered as a distinctive algorithm considering to metrics aspect. Ideally, in order to have trustworthy benchmarking among different approaches, it is necessary to have a flexible implementation that can be used by everyone regardless the dataset used. As an object detector the metrics that are responsible to evaluate how well operates are true and false positive, respectively and as well true and false negative which are based on the IOU algorithm. To understand better, let's make the following definitions:

- **True Positive (TP):** A correct detection, Detection with IOU $\geq$ threshold
- **False Positive (FP):** A false detection, Detection with IOU $<$ threshold
- **False Negative (FN):** A ground truth not detected.
- **True Negative (TN):** Does not apply. It would represent a corrected misdetection. In the object detection task there are many possible bounding boxes that should not be detected within an image. Thus TN would be all possible bounding boxes that were correctly not detected (so many possible boxes within an image). That is the reason, why it is not used by the metrics.
- **Precision:** Is the ability of a model to identify only the relevant objects.

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

- **Recall:** Is the ability of a model to find all the relevant cases (all ground truth bounding boxes).

$$\boldsymbol{Recall = \frac{TP}{TP + FN} = \frac{TP}{all\ ground\ truth\ boxes}}$$

As described in *section 2.3*, Yolo algorithm's output consists of 13x13 plus 26x26 grid outputs, which means that the architecture extracts 2.535 predicted boxes, from 845 cells. Taking into consideration two major facts that, first and foremost YoloV3-Tiny is a multi-class network, as a result there is the need to calculate precision and recall so many times as the number of classes are, and secondly, some cells could be no responsible for a detection of an object making them inactive and out of charge, creating the need of conforming to the cells which are responsible to detect the target object. To clarify the primary building blocks of the metrics, we need to inherit the IOU algorithm. In those cells which are responsible for detection, network extracts 3 bounding boxes per cell with the encoding of object probability, box coordinates, and class probabilities. By applying the IOU algorithm with the target box coordinates and these 3 bounding boxes coordinates we take 3 IOU scores per bounding box, but holding only the maximum IOU score among these 3. As a result, each responsible cell is represented by a bounding box, solely, with its object probability and its class probabilities. Also, to each responsible cell has been assigned a specific class as a target. To calculate the precision of the network, we need the true and false positives for each class. IOU is measured between predicted and target boxes. We measured the true positives as the number of those bounding boxes in which the maximum IOU is greater than the specified threshold (0.5), and false positives as the number of those bounding boxes in which the maximum IOU is less than the threshold [4].

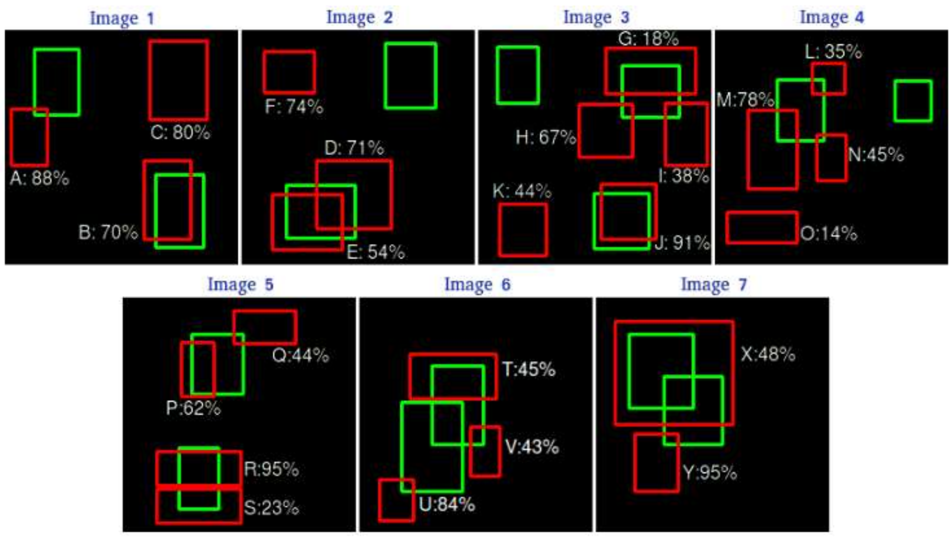


Figure 2.15: There are 7 images with 15 ground truth objects represented by the green bounding boxes and 24 detected objects represented by the red bounding boxes. Each detected object has a confidence level and is identified by a letter (A,B,...,Y).

Let's make an example and measure the precision of the predictions of the figure 2.15. Let's assume that the IOU threshold is 0.3. In some images there are more than one detection overlapping a ground truth. For these cases the first detection is considered TP while the other are FP [4].

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

Figure 2.16: Measurement TP and FP for the predictions of the figure 2.15.

$$Precision = \frac{7}{24} = 29.17\%, \qquad Recall = \frac{7}{15} = 46.66\%$$

2.7. Multi-Class and Multi-Label Classification

In Yolo algorithm multi-class classification is implemented by passing through the sigmoid (figure 2.13 equation 6) the predicted classes of each bounding box and multiplied them by the predicted probability of object existence. The multiplication's product is called confidence score probability and as a consequence, the maximum probability among these is the predicted class.

$\underline{cl}$ is encoded as a one-hot probability vector and p_c as probability object's existence:

$$cl = \big(p(fire), \quad p(smoke)\big), \quad p_c = p(object),$$

$$where\ p\ defines\ the\ probability$$

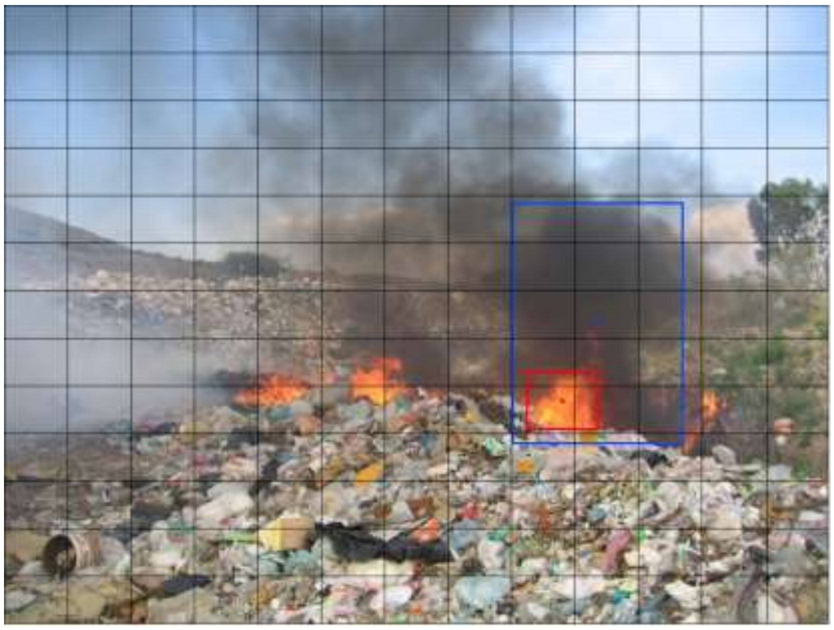
Confidence score probability defined as the conditional probability of each class given the fact that there is an object.

$$p(cl|object) = (\mathrm{p_c} * \mathrm{p(fire)}, \quad \mathrm{p_c} * \mathrm{p(smoke)})$$

$$p(cl|object) = (\mathrm{p(fire|object)}, \quad \mathrm{p(smoke|object)})$$

Given that, confidence score probability is responsible for detection. Then, it crosses the threshold of Non-Max Suppression algorithm, as a consequence to filter out the unnecessary boxes.

For example: Let's assume, two predicted bounding boxes with the following values as shown in figure 2.17b.



(a)

Output	b_x	b_y	b_w	b_h	p_c	cl	Confidence Score probability (fire, smoke)
bbox1	9.34	6.66	216	307	0.88	(0.59,0.95)	(0.5192, 0.836)
bbox2	8.78	8.29	90	76	0.93	(0.96, 0.51)	(0.8928, 0.4743)

(b)

Figure 2.17: (a) Predicted Output with drawing 13x13 grid, (b) two bounding boxes responsible for detecting fire (red box) and smoke (blue box)

The network has already predicted and filtered out the unnecessary bounding boxes holding the 2 bounding boxes as shown in the figure 2.17a and 2.17b. The confidence score probability define which class has been predicted by bounding box, corresponding to which value is the greater. The integer part of b_x, b_y shown which cell was responsible for the detection of specific bounding box and the rest part shown the percentage of how much it has been shifted from the top-left point of the specific cell. The b_w, b_h define the size of width and height.

Chapter 3

Tensorflow – Tensorflow Lite

3.1. Introduction to TensorFlow

Tensorflow is the successor to DistBelief, which is the distributed system for training neural networks that Google has used since 2011 [5]. It's an end-to-end source platform for machine learning systems and it operates at large scale and in heterogeneous environments [6]. Additional to that, it's considered as a flexible interface for expressing machine learning algorithms and implementations for algorithms execution, like training and inference algorithms for deep neural network models. Moreover, Tensorflow uses dataflow graphs to represent computation, shared state, and the operations that mutate that state.

At the most traditional dataflow systems, graphs vertices represent functional computation on immutable data but, Tensorflow allows vertices to represent computations that own mutable state. Edges carry tensors (multi-dimensional arrays) between nodes and Tensorflow inserts the appropriate communication between distributed sub-computations on the fly. A computation can be executed with subtle changes on a wide variety of heterogeneous systems, ranging from mobile devices and embedded systems such as phones and tablets up to large-scale distributed systems of many machines and thousands of computational devices such as Graphical Processing Units (GPUs). As a result, it has been used for a variety of research fields, including computer vision, robotics, speech recognition information extraction, even research and computational in medical discipline, and drug discovery.

3.2. Tensorflow execution and dataflow graph.

Tensorflow uses a single dataflow graph to represent all computation and state in a machine learning algorithm, including the individual mathematical operations, the parameters and their update rules and the input preprocessing (figure 3.1). The communication is explicitly expressed between sub-computations by dataflow graph making it easy to execute independent computations in parallel [5]. Tensorflow differ from batch dataflow systems in two respects:

- The model supports multiple concurrent executions on overlapping subgraphs of the overall graph.
- Individual vertices may have mutable state that can be shared between different executions of the graph.

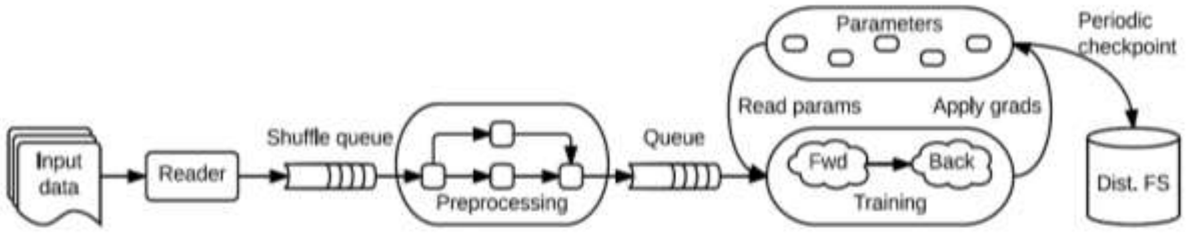


Figure 3.1: A schematic Tensorflow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpoints state.

In the parameter server architecture, the secret about observation is, that mutable state is crucial when very large models are trained, because it allows to make in-place updates to very large parameters, and propagate those updates to parallel training steps as quickly as possible. Dataflow with mutable state enables Tensorflow to mimic the functionality of a parameter server, but with additional flexibility.

Each vertex, in a Tensorflow graph, acts on the behalf of a unit of local computation, and each edge the output from, or input to, a vertex. The reference to operations is the computation at vertices and to the tensors is the values flow along edges.

Tensors: In Tensorflow, tensors are all data with n-dimensional arrays having each element one of primitives types, such as int-32, float-32m or string. Moreover, tensors naturally represent the inputs and outputs-results of the common mathematical operations in machine learning algorithms.

Operations: The procedure, which take m >= 0 tensors as input and produces n >= 0 tensors as output, is called an operation. An operation may have zero or more compile-time attributes determining its behavior.

3.3. Tensorflow-Eager

Tensorflow Eager is a Python-embedded DSL (Domain-Specific Language) for differentiable programming that lets developers interpolate between imperative and staged computations in a single package [7]. Additionally, it offers a multi-stage programming model, allowing a rapidly prototyping method for programs and selectively stage parts that need to accelerate or serialize. It provides an imperative execution model which immediately executes their kernel without the need of conversion to a dataflow graph. It is implemented as an opt-in extension to Tensorflow.

While imperative execution simplifies prototyping, the overhead of going back and forth to the Python interpreter creates a bottleneck, limiting its performance. To take advantage of the benefits of dataflow graphs, Tensorflow Eager provides a Python decorator that traces its Python function in a graph-building context, staging primitive operations to construct a dataflow graph with named inputs and outputs. The execution of graph functions bypasses Python while they are executed using a C++ dataflow runtime or compiled to generate optimized code for CPUs, GPUs, and ASICs. A lexical environment is shared by graph functions and imperative code, as a result to make it simple to invoke graph functions for imperative code, create graph functions, and embed imperative code in graph functions via unstaging annotations. To sum up, Tensorflow Eager provides two ways of executing operations: imperatively or as part of static dataflow graph. Tensorflow is able to produce and run dataflow graphs, and computations easily, in a variety of heterogeneous systems, that’s why Tensorflow Lite plays a vital role to embedded systems and mobile phones.

3.4. Tensorflow Lite

Tensorflow Lite is the official cross-platform framework to run with models on edge devices, mobile devices, and embedded systems, and it's deployed in billions edge devices around the world supporting Android, IOS, Linux-based IoT devices and microcontrollers. Tensorflow Lite is able to convert Tensorflow models, which have been already trained, and make them to run in systems with memory limitations, more power efficiently with great performance, considering the size benefits of quantization and this kind of method is called post-training quantization. Moreover, there is also a method to drive into best converted weights and it is called quantization-aware training.

3.4.1. Quantization

Quantization is the process of transforming an machine learning model into an equivalent representation that uses parameters and computations at a lower precision. This improves the model's execution performance and efficiency. Tensorflow Lite is targeting to convert models based on float precision to models based on float with lower precision or even on integer precision. For example, 8-bit integer quantization results in models which are up to 4 times smaller in aspect of size, 1.5-4 times faster in computations, and lower power consumption on CPUs. Furthermore, it provides model execution in specialized neural accelerators, such as EdgeTPUs in Coral, which has a restricted set of data types. However, it's easily comprehensible that, the process of going from higher to lower precision is lossy in nature.

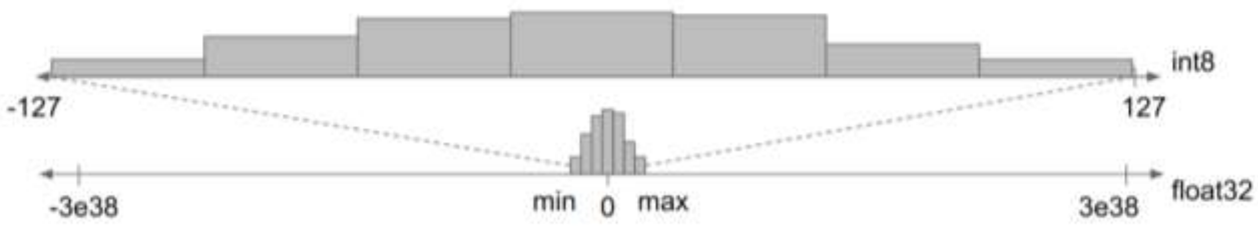


Figure 3.2: Small range of float-32 values mapped to int-8 is a lossy conversion since int-8 has only 255 information channels.

As shown in the figure 3.2 above, quantization is the conversion of a small range of floating point values into a fixed number of information buckets. The parameters of a converted model can take a small set of values and the detailed differences between them are lost. We could say that this is similar rounding errors when fractional values are represented as integers.

Current quantization approaches are split to two categories. The 1st category exemplified by MobileNet, SqueezeNet, ShuffleNet, and DenseNet designs that exploit computation / memory efficient operations. Secondly, the next category quantizes the weights or activations of a Convolutional Neural Network from 32-bit floating point into lower bit-depth (quantization). Quantization is lacking in aspect of trading off latency with accuracy [8].

1. First approach is related to models, which are over-parameterized by design in order to extract marginal accuracy improvements. However, reducing quantization experiments of these architectures results in an easy compression and proof-of-concepts at best. Instead, a more meaningful challenge would be to quantize model architectures that are already efficient at trading off latency with accuracy.
2. Secondly, many quantization approaches do not convert efficiently and verifiably models on real hardware. Approaches that quantize only the weights are to deal with on-device storage and less with computational efficiency.

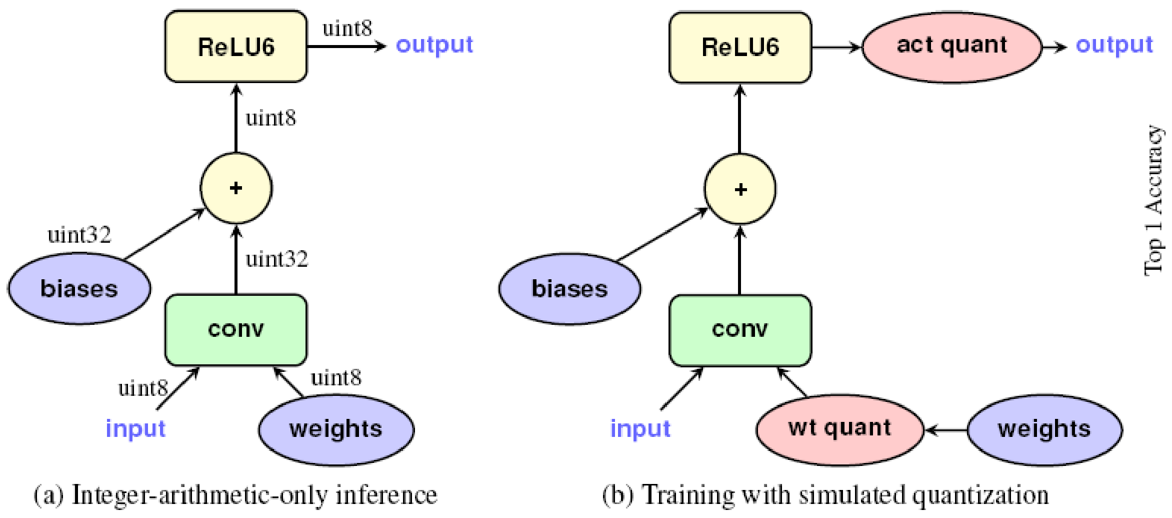


Figure 3.3: a) Integer-arithmetic-only-quantization inference of a convolution layer. b) Training and inference with simulated quantization of the convolution layer. All variables and computations are carried out using 32-bit floating-point arithmetic. Weight quantization and activation quantization nodes are injected into the computation graph to simulate the effects of quantization of the variables.

In integer post-training quantization doesn't require any modification to the network, only in precision of the weights (as shown in figure 3.3a) instead of quantization-aware training. It uses "fake" quantization nodes in the neural network graph (as shown in figure 3.3b) to simulate the effect of the 8-bit values during training. Thus, this technique modification to the network before initial training, forbidding transfer learning. This generally results in a higher accuracy model because it makes the model more tolerant of lower precision values, due to the fact that the 8-bit weights are learned through training rather than being converted later. The issue with this method is the compatibility among the different version of Tensorflow, as for Tensorflow 2.0 and newer versions there is not compatibility.

3.4.2. Arithmetic Conversion for Quantization and Batch Normalization folding

Quantization logic has a basic requirement which permits efficient implementation of all arithmetic precisions using only integer arithmetic operations on the quantized values (as paper [8] refers to no use of lookup tables, because these tend to perform poorly compared to pure arithmetic on SIMD hardware). The affine mapping and conversion of quantization technique is:

- $r = S(q - Z),\ where\ S\ is\ scale, Z\ zero\ point, r\ real\ number, and\ q\ integer$

NOTE: The equation above plays a major role in the implementation of the YoloV3-Tiny, during inference in the Google Coral Dev Board.

The Zero-point (Z) is in fact the quantized value q corresponding to the real value 0, it could be considered as shifting from the 0 real number to quantized integer number. The Scale (S) is an arbitrary positive real number, which is represented in software as a floating-point quantity. In paper [8] are described methods for avoiding the representation of such floating-point quantities in the inference workload. Furthermore, another important issue for the quantization of YoloV3-Tiny, and it is worth noting is Batch Normalization folding. The inference graph has batch normalization parameters folded into the convolutional or the fully connected layer's weights and biases, for efficiency. To accurately simulate quantization effects, the folding of batch normalization parameters is need to be simulated in order to scale the weights. The equation to scale them is:

- $w_{fold} := \frac{\gamma w}{\sqrt{\sigma_B^2 + \varepsilon}}$

After folding, the batch-normalized convolutional layer reduces to the simple convolutional layer as shown in the figure 3.3b.

3.5. Colab by Google

Colab is a free, browser-based notebook environment that is able to entirely run code in the cloud. It provides massive power by using either its processor or its GPU or a TPU. First and foremost, it needs to decide in which environment you would like to run, considering which accelerator you prefer (None->Use CPU only, GPU, TPU). Colab provides a range of GPUs among NVIDIA K80, P100, P4, T4, V100.

The training of the network's weights was based on GPUs, the K80, P100, and Tesla T4 speeding up a lot the procedure of convergence.

- K80 is able to provide 1.87, and 5.6 teraFLOPS of single and double precision performance, respectively, in conjunction with the high memory capacity which is provide 12GB of RAM [9].
- P100 benefits from NVIDIA Pascal architecture delivering a superior performance with more than 21 teraFLOPS of 16-bit floating-point. Moreover, it delivers over 5 and 10 teraFLOPS of double and single precision for high performance computing, in association with 16 GB of memory [10].
- T4 is the best among these that used for training. It provides 320 NVIDIA Turing Tensor Cores, delivering revolutionary multi-precision performance with 65 teraFLOPS, especially in machine learning. It provides an extraordinary performance with more than 8.1 teraFLOPS in single-precision [11].

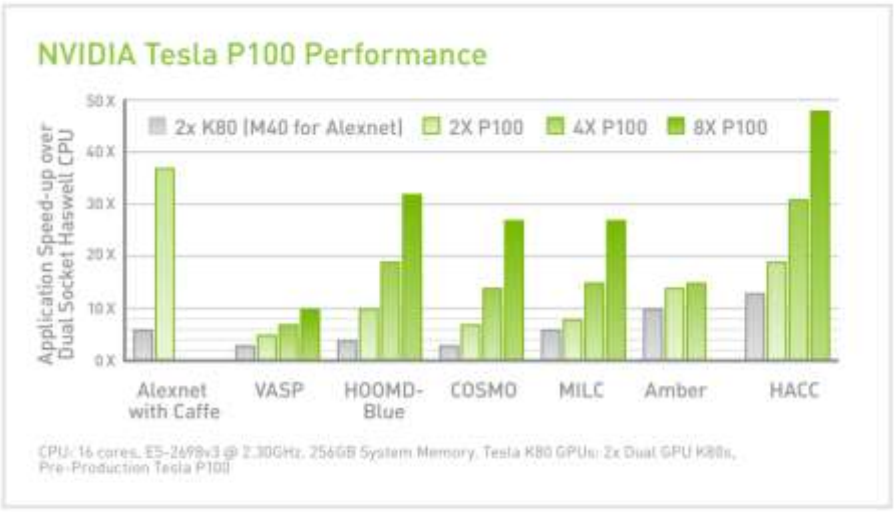


Figure 3.4: A chart table comparing the performance of K80, P100, providing up to 2x, 4x, 8x of each GPU. The chart table is shown as given by [10]

On the other hand, the TPU is available on Colab is TPU-V2 with 8-cores, and each core provides 8GB of memory. In total, the TPU is able to use 64GB of high Bandwidth Memory.

The most import note about TPU is that is capable to process with the speed of 180 teraFLOPS!

The negative issue about the use of Colab is the 12-hour straightforward timeout, making urgent the need of rapid training and transfer learning. Moreover, if the time passed using the GPU, then there is a 12-hour ban from using this kind of accelerator.

Chapter 4
Target platform

4.1. Coral

Coral is a Google-Research department which have designs on bringing on-device AI application ideas from prototype to production. It provides a complete platform for accelerating neural networks on embedded devices. Coral is a hardware and software platform for building intelligent devices with fast neural network inferencing. The heart of the Coral devices is based on the Edge TPU coprocessor, which is a tiny ASIC module built by Google that’s specially designed to execute state-of-the-art neural networks at high speed, with low power cost driving into power efficient devices.

4.1.1. Google Coral Dev Board

Coral provides a variety of different hardware components, but Google Coral Dev Board is a single-board computer that’s ideal solution for fast machine learning inference in a small factor. It relies upon the i.MX8M SoC consisting of a quad-core Arm Cortex-A53 MPCore platform and an Arm Cortex-M4F platform, eMMC memory, LPDDR4 RAM, Wi-Fi, and Bluetooth, but its special and unique power comes from Google’s EdgeTPU coprocessor. Also it provides only 8MB of embedded RAM in the Edge TPU, limiting the model’s size.

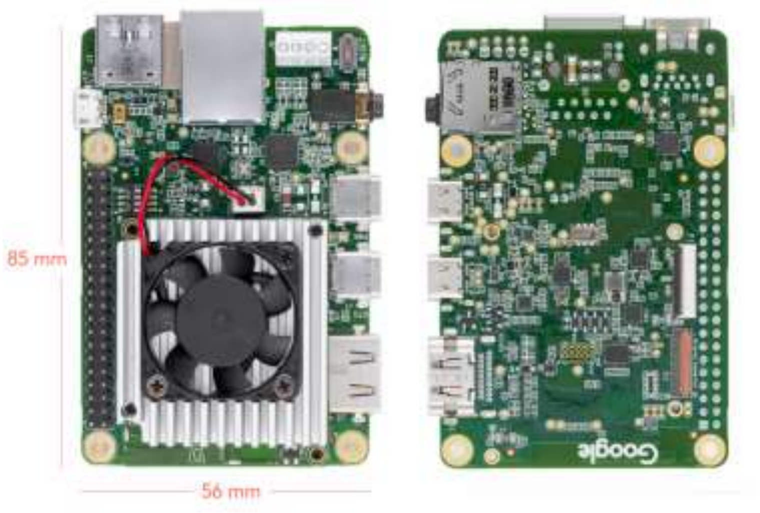


Figure 4.1: Google Coral Dev Board schematic.

4.1.2. Edge TPU

The Edge TPU is capable of performing 4 trillion operations (Tera-Operations) per second (4 TOPS), using 0.5 watts for each TOPS (2TOPS per watt). The following figure compares the inference time for several popular vision models in Tensorflow format, when executed on the embedded CPU or on the Coral Dev Board. Edge TPU is based on the normal TPU with systolic arrays, but there is no published paper for its architecture

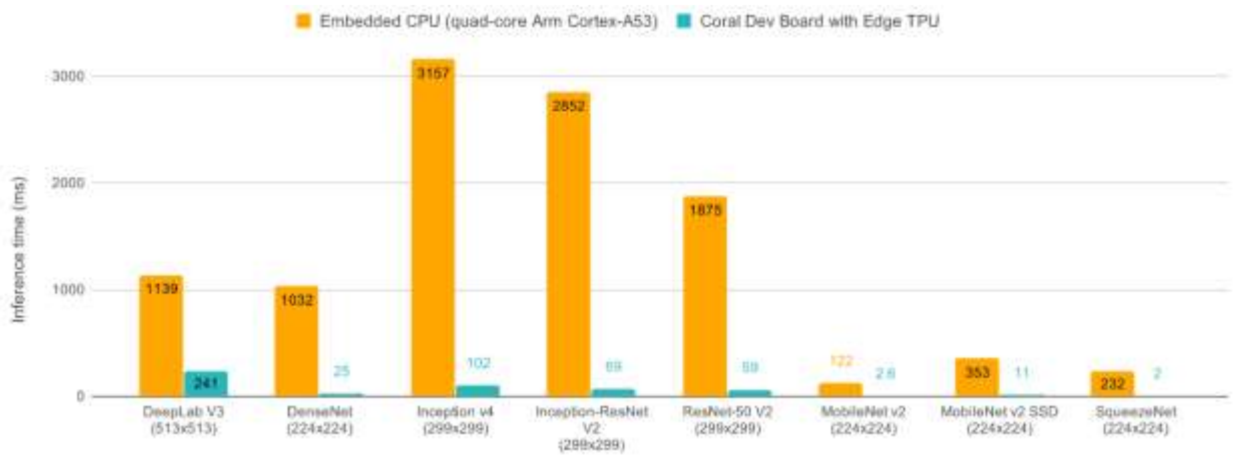


Figure 4.2: Comparison of inference time among a variety of models (lower is better).

4.1.3. Mendel Linux

Coral Dev Board is operation system is based on Linux and it is a derivative of Debian Linux called Mendel. It has been optimized for embedded systems by making it very lightweight. Mendel includes also the required tools to build headless machine learning applications, using Python and C++ libraries, the Edge TPU API, the Tensorflow Lite API, and the Edge TPU runtime. Moreover, there is a tool called MDT (Mendel Development Tool) which it can easily connect and securely transfer files or run other commands from a remote computer to the device.

4.2. Model Compatibility Overview

In order for the Edge TPU to provide high-speed neural network performance in conjunction with a low-power cost, the Edge TPU supports a specific set of neural network operations and architectures. The Edge TPU is capable of executing deep feed-forward neural networks such as Convolutional operations. It supports only Tensorflow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU. It achieves low-latency inference in a small binary size – both Tensorflow Lite models and interpreter kernels are much smaller.

In order to take advantage of the Edge TPU at runtime, Tensorflow model must meet the following requirements:

- Tensor parameters are quantized (8-bit fixed-point numbers. Either int-8 or uint8).
- Tensor size are constant at compile-time (no dynamic sizes).
- Model parameters are constant at compile-time.
- Tensors are either 1-,2- or 3-dimensional. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- The model uses only the operations supported by the Edge TPU [EDGE-TPU operations]

Some of the EDGE-TPU operations:

- ResizeNearestNeighbor: Depending on input/output size, this operation might not be mapped to the Edge TPU to avoid loss in precision,
- Conv2D,
- ReLU, but not LeakyReLU
- Reshape: Certain might not be mapped for large tensor sizes.

4.3. Edge TPU Compiler

The Edge TPU Compiler is a command line tool that compiles a Tensorflow Lite model into a file compatible with Edge TPU. The Edge TPU has roughly 8MB of SRAM that can cache the model's parameter data. However, a small part of the RAM is first reserved for the model's inference executable, so the parameter data uses whatever space remains after that. Naturally, storing the parameter data on the RAM of the Edge TPU enables faster inference speed compared to fetching the parameter data from external memory.

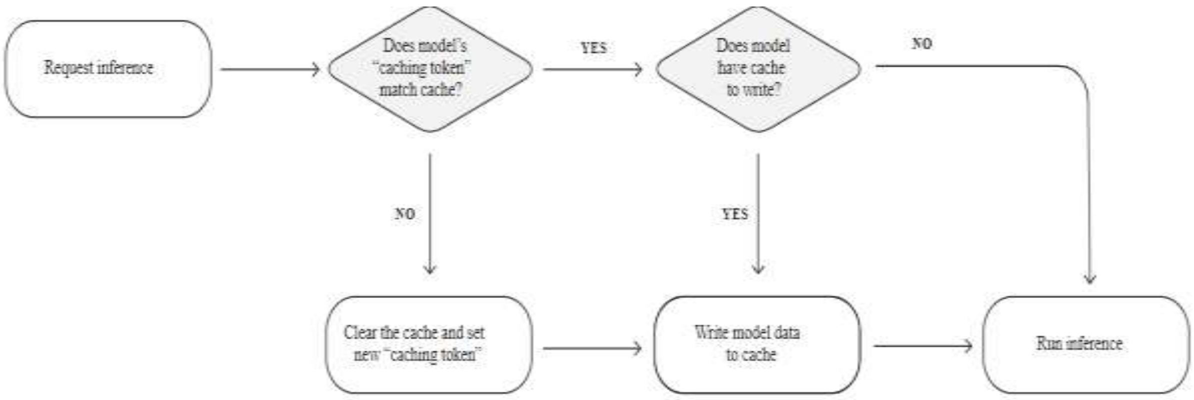

Request inference
Does model's "caching token" match cache?
YES
Does model have cache to write?
NO
NO
YES
Clear the cache and set new "caching token"
Write model data to cache
Run inference

Figure 4.3: Flowchart how the Edge TPU runtime manages model cache.

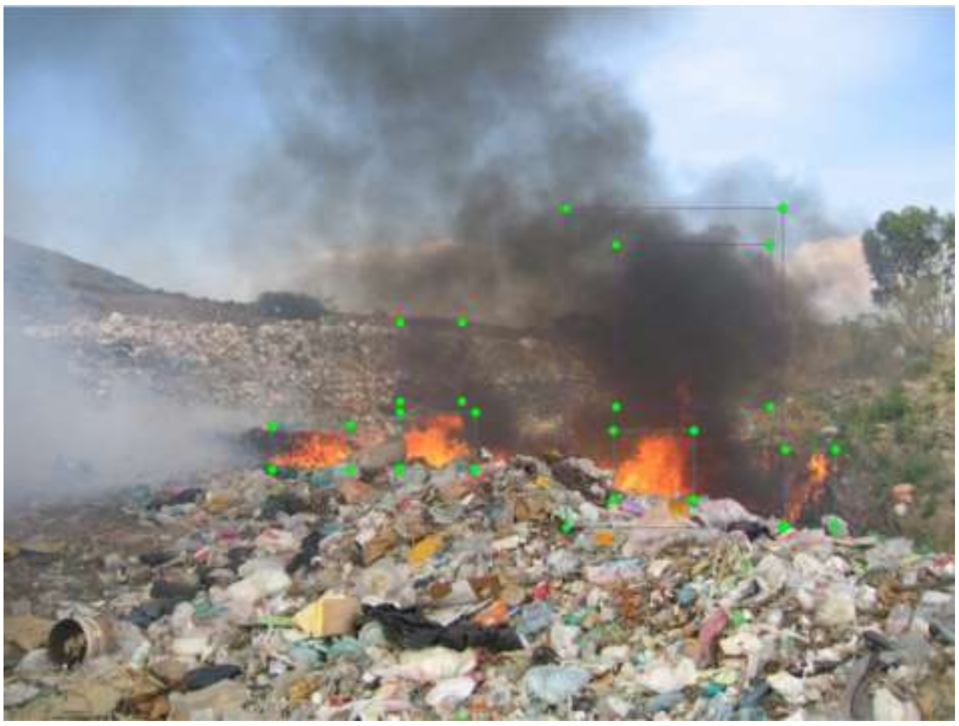
The Edge TPU "cache" is not actually traditional cache, but it is a compiler allocated scratchpad memory. The Edge TPU Compiler adds a small executable inside the model that writes a specific amount of the model's parameter data to the SRAM (if available) before running an inference. When models are compiled individually, the compiler gives each model a unique caching token encoded from 64-bit number. Then, in inference time the Edge TPU runtime compares that caching token to the token of the data that is currently cached. If tokens matched, the runtime uses that cached data. If they don't match, it wipes the cache and writes the new model's data. This process logic is shown in figure 4.3.

Chapter 5

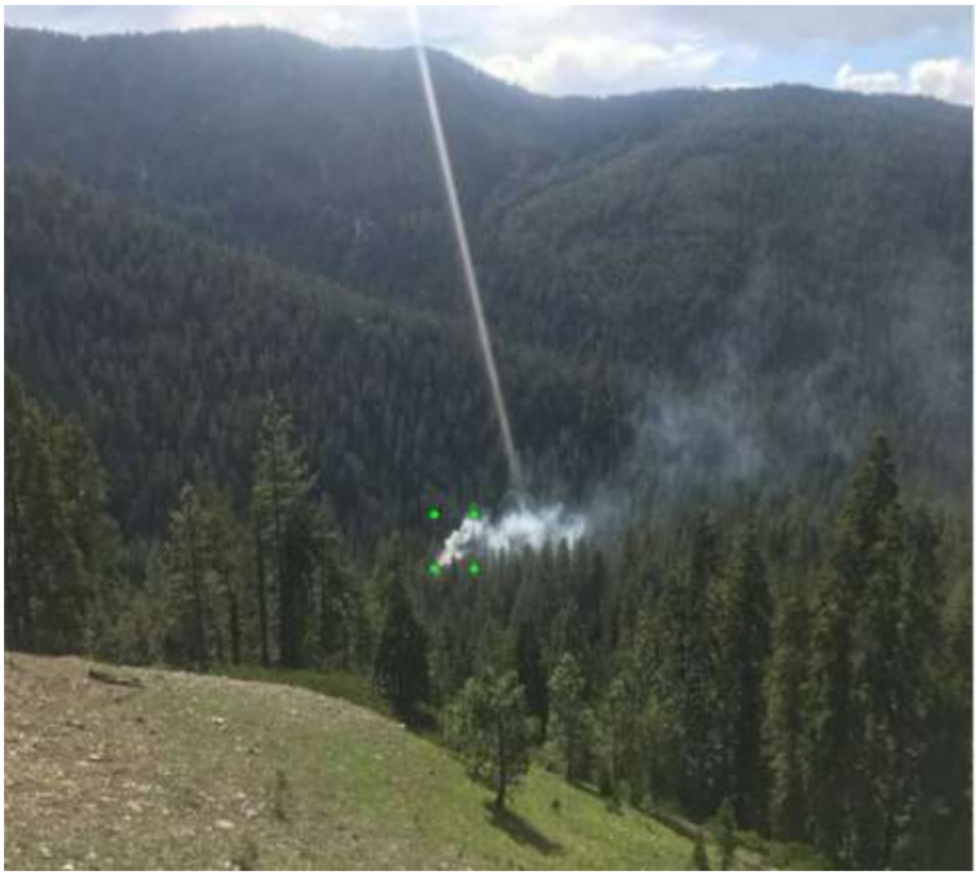
Fire detection using EdgeTPU

5.1. Data gathering

To implement a project using deep neural network detection the most vital issue is the collection of data, in project’s case by referring to data it is meant images, consisting of objects that it is going to detect. A part of the dataset has been collected by searching in the internet for photos which depict wildfire or the start of this, with only a tiny smoke column or a small amount of fire. The rest of the dataset has been collected, after I had communicated with two local fire departments of Greece, in Volos and Halkida, to verify, whether they are willing to help and share with me photos or video (in order to extract frames of it) or not, which they were captured when the fire and smoke state was at an early stage, and before the scene evolved into wildfire, and calamity. They got excited by the idea, but they inform me that the solely data that they could have found an share with me was, data from scenes where the wildfire was got ease, or even data in which the depicted places were totally burned. Then, my supervisor Mr. Bellas Nikolaos and I decided to write a formal document in order to send a request to the center facility of fire department of Greece, but unfortunately, we didn’t get any response. After that fact, we decided to go on with the data that we had already collected from the Internet and Google Search.



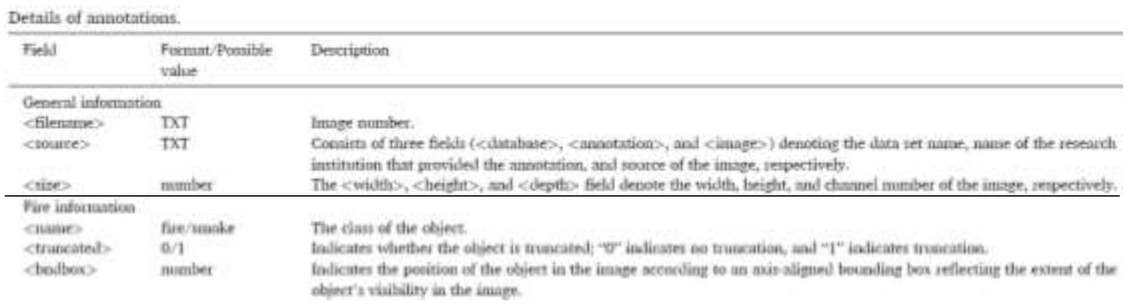
(a)



(b)

Figure 5.1: Annotated images samples from dataset. (a) Fire started in a rubbish place. (b) Smoke in a forest.

The Data consists of a gathering of 160 images, they have been annotated with labelImg tool [19] drawing the boxes which they would be the target of the network. LabelImg saves all the information about annotation in a xml file format, like in figure 5.2.



Details of annotations.

Field	Format/Possible value	Description
General information		
<filename>	TXT	Image number.
<source>	TXT	Consists of three fields (<database>, <annotation>, and <image>) denoting the data set name, name of the research institution that provided the annotation, and source of the image, respectively.
<size>	number	The <width>, <height>, and <depth> field denote the width, height, and channel number of the image, respectively.
Fire information		
<name>	fire/smoke	The class of the object.
<truncated>	0/1	Indicates whether the object is truncated; "0" indicates no truncation, and "1" indicates truncation.
<bndbox>	number	Indicates the position of the object in the image according to an axis-aligned bounding box reflecting the extent of the object's visibility in the image.

Figure 5.2: Ground truth Boxes annotation.

The next step is to cluster the aspect ratios (width and height) of target bounding boxes that labelImg has exported by applying the K-means algorithm in order to extract 6 anchor boxes (figure 5.3), in turn, they are going to anchor to the corresponding network's output, forming the predicted final bounding box.

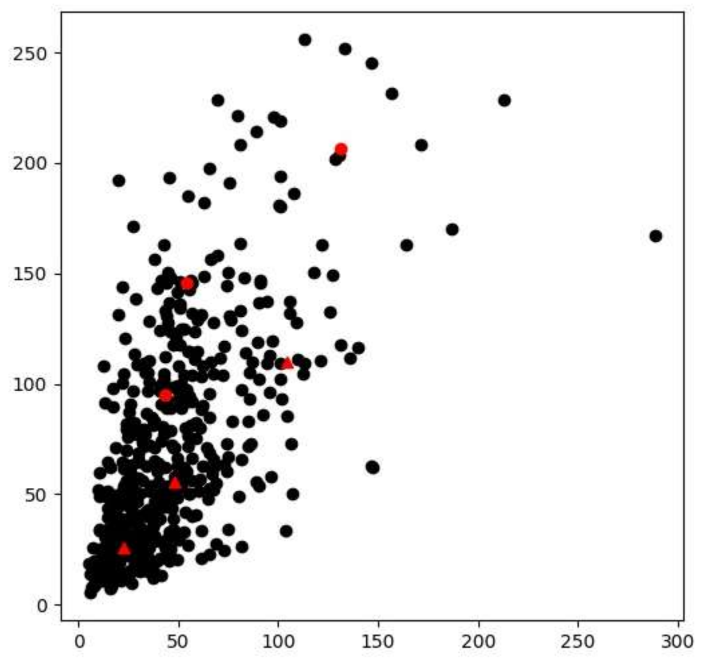


Figure 5.3: Aspect ratio variance of the Dataset. The colored points are the 6 anchors aspect ratios.

According to small size, the dataset has been split into train-set and validation-set in a percentage of 70%, 30%, respectively. The dataset must be prepared to be assigned into the model to begin and train its weights. To read data efficiently it can be helpful to serialize your data and store it in a set of files (100-200MB each) that each can be read linearly. This also can be useful for caching any data-preprocessing and make even quicker the training. Tensorflow gives the advantage to prepare easily this kind of dataset based on fundamental operation that it has. *Tensorflow.train.Feature()* operation parses an xml file taking the information that they have been assigned to it, in order to give them to *tensorflow.train.Example()* which, in its turn, builds a Protocol Message and finally pass it to *tf.io.TFRecordWriter()* extracting a TFRecord format file for storing a sequence of binary records. The last operation uses the Protocol Buffers which are a cross-platform, cross-language library for efficient serialization of structured data developed by Google.

5.2. Network Fitting

The method of architecture’s training is a vital part of how well the network is going to operate. Usually, the most preferred method is to transfer the weights values of an architecture that has been already trained, which is called transfer learning. As each person learns to detect a new object by observing it a few times and he does not need to learn how to see from the beginning. Generally, he needs only to transfer their knowledge of learnt domains to newer domains and tasks [18]. Transfer learning is the best and most rapid way to train a network in order to converge, instead of random initialization of weights and training them from scratch. To have a desirable effect of transfer learning, it is proposed a 2-stage strategy, as a consequence of having a small amount of data. The key idea for the proposed training strategy is just to leverage the pre-trained architecture weights layers to extract features but not to update the weights of the model’s layers during training.

The pre-trained weights are derived from the coco dataset and downloaded from Joseph Chet Redmon website in order to be read and be converted into weights that are capable to load on a tensorflow model.

Moreover, the framework that was used for building, training, running the model is the Tensorflow in the version 2.2 (*section 3*), which is an easy to use framework but with compatibility issues for converting in tensorflow lite model. This kind of issue will be explained later on.

5.2.1. Model Compilation

The compilation of the model is inclusive of the loss, optimizer, and metrics assignments. For each output has been assigned a loss function, as it has been already mentioned in *section 2.5*, according to anchor boxes area which they have been sorted in descending order, and they have been assigned 3 anchor boxes per output (6 in total).

Adam has been chosen as optimizer in order to efficiently and rapidly update the weights of the network. Adam's name is derived from adaptive moment estimation, and it is a method for efficient stochastic optimization that only needs first-order gradients with little memory requirements, making the training time even smaller. It computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. Some of the Adam's advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient of the gradient, its learning rates are approximately bounded by the original learning rate hyperparameter, it does not require a stationary objective, it naturally performs a form of step size annealing [12].

Moreover, the compilation needs to load the metrics that are going to evaluate the model in order to have more completed view of how well is the model‘s operation. Metrics that is used it has been described in *section 2.6*

5.2.2. Training Strategy and Performance

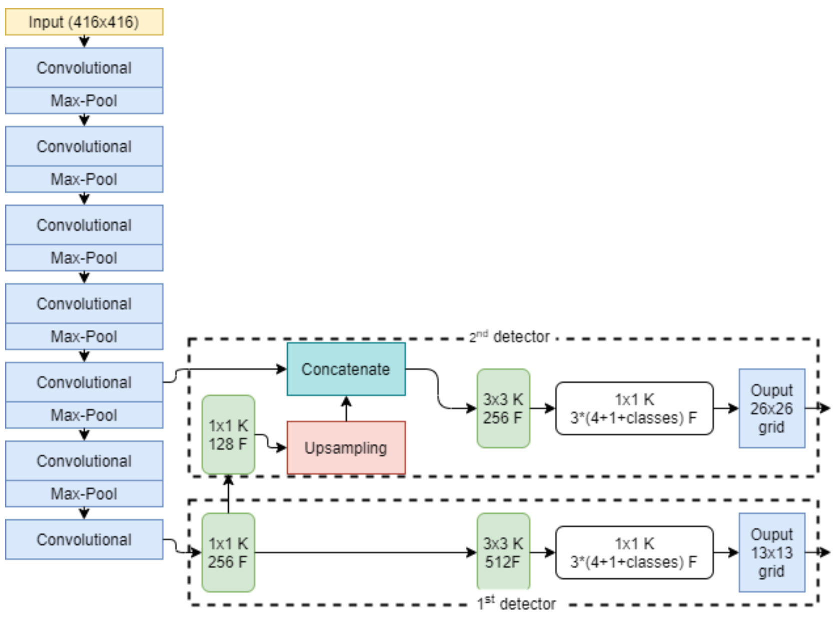
As I mentioned in previous (*section 2.2*), Yolo architecture is separated in two main components Feature Extractor and Detectors, as a consequence in two main categories [14].

The training strategy of the 1st stage (*stage-1*) it freezes the parameters of the Feature Extractor and re-adjust the parameters of the Detectors, since I would hardly improve the ability of the base model to extract simple shapes, such as diagonals or gradients (frozen layers of the architecture are shown in figure 5.4a). Moreover, the parameters of Detectors are parameterized by the number of classes that it is going to detect as it is shown in figure 2.10. As a consequence, the need of freezing the Feature Extractor's weights and train only the weights of Detectors from scratch it is extremely crucial for the convergence of the network. An appropriate learning rate (10^{-3}) has been chosen for the adjustment, not too high to prevent oscillations, nor too low avoiding long time waiting. Training time is small because there are not many parameters to tune, a few epochs.

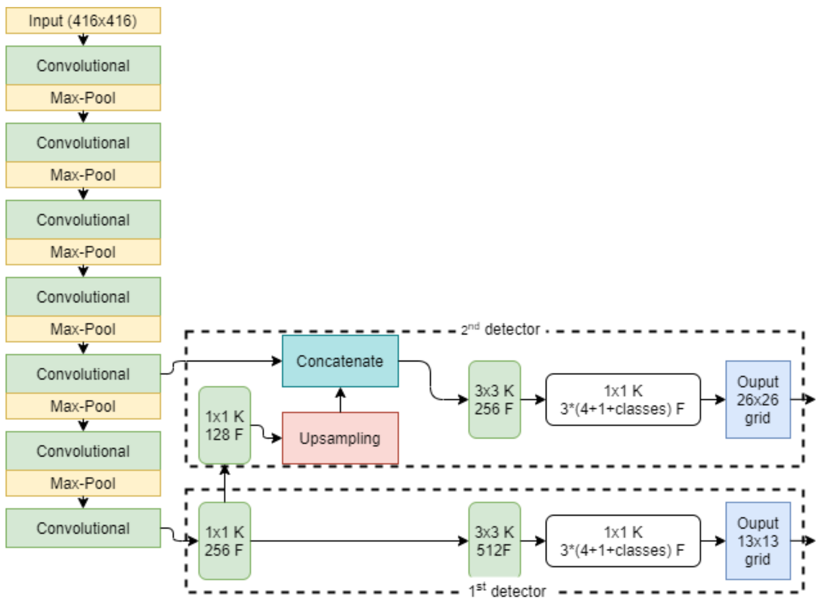
In the 2nd stage (*stage-2*), the parameters of the Feature Extractor are unfroze and a fine tune of all network weights is going to take effect (active layers of the architecture are shown in figure 5.4b). However, the range of learning rate among of architecture layers barely change, while the final weights have a greater modification margin than those of first layers, it is need to use a smaller learning rate (10^{-4}) for more epochs.

This flow training strategy steps could be repeated as many times as the developer thinks it is needed but, the flow must end up with stage-1 in order to converge the weights of the final layers according to the updated weights of the Feature Extractor.

Note: The Tensorflow stores the weights which have the least possible loss according to the summation of two outputs during evaluation of validation-set. Moreover, as illustrated in both images of figure 5.4 during training the architecture does not contain the Non-Max Suppression. Because this Layer is used for crossing out multiple valid detections of the same object.

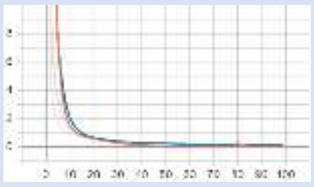
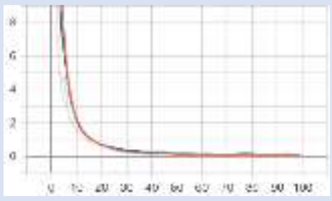
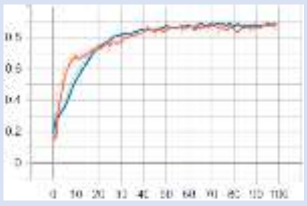
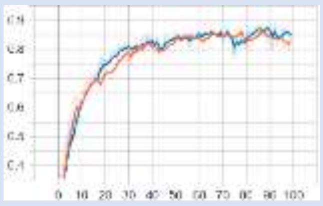


(a)

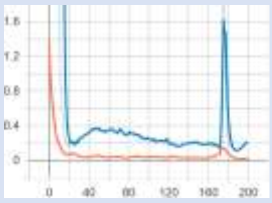
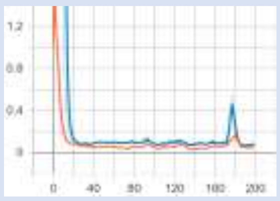
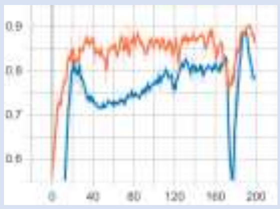
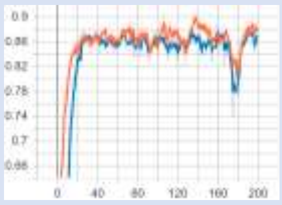


(b)

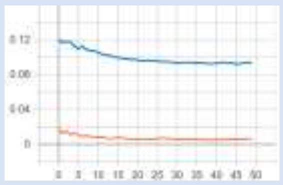
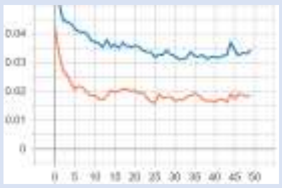
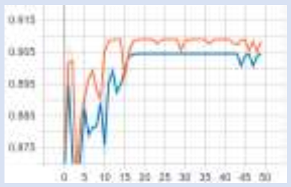
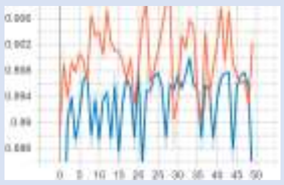
Figure 5.4: (a) The YoloV3-Tiny architecture during Transfer Learning with Feature Extractor parameters frozen. stage-1. (b) The YoloV3-Tiny architecture during Transfer Learning with Feature Extractor parameters unfrozen. stage-2.

Stage-1	Output 0 (13x13)	Output 1 (26x26)
Loss		
Precision		

(a)

Stage-2	Output 0 (13x13)	Output 1 (26x26)
Loss		
Precision		

(b)

Repeated Stage-1	Output 0 (13x13)	Output 1 (26x26)
Loss		
Precision		

(c)

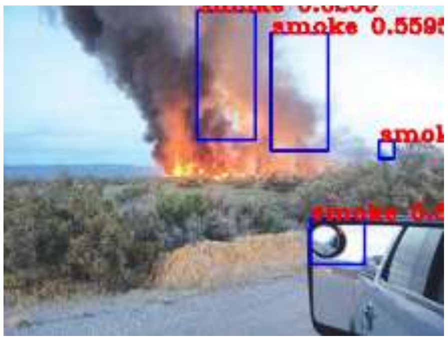
Figure 5.5: For each chart 1st row depict the loss update among the epochs and the 2nd row depict the precision ascent. The orange plot is the updated values from train-set, and the blue one is from validation-set. (a) Extracted images during the 1st stage of the training, (b) during the 2nd stage of the training and (c) the 1st stage is repeated for better convergence based on the updated Backbone's weights.

Tensorflow provides very interesting tools for training and analyzing the model's operation. Tensorflow has important tools once fitting the model, making it, capable to store the weights that has the better performance (metrics) and lower loss, based on the results of the validation-set.

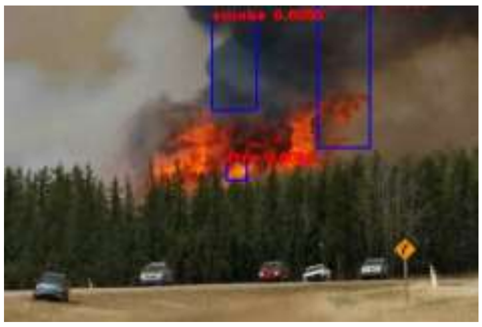
To analyze better the performance of the network figure 5.5 consists of 3 tables with 4 charts each. These charts were exported by Tensorboard, a Tensorflow's tool. It provides metrics information for each stage of the training strategy, as explained. In figure 5.5 the orange plots are based on the train-set values and these with the blue color are based on the validation-set. During training the weights were being stored, were those which had the least loss value. So, when we refer to the precision scores for each stage of the training are the scores in which the validation loss was the minimum.

During training of the weights and following the strategy described above, the transfer learning take effect extremely efficiently as we can see In figure 5.5a. The loss sharply sloped downwards, and it almost goes down to zero, resulting in a great fitting of the last layers' weights to the dataset, and causing in its turn, a steep ascent to the precision of the network. The reason of the great slopes is that the Feature Extractor is able to extract fundamental features capable to detect object like fire and smoke. The precision scores were achieved, were 89.29% and 88.89% for each output. The next step was to re-train the whole network's weights and make a subtle adjustment of the extracted features with a small learning rate, not making a better change but Feature Extractor training helps to update its weights for more fundamental features of the specific dataset. This is the reason, why we see a slightly rise of loss and fall of precision. (figure 5.5b). At the last step of the training, the Feature Extractor was frozen and the Detectors are trained according to the new updated values of the Backbone's weights. At the last step, the updates weights improved the precision at 90.05% and 90.02% for each output, respectively (figure 5.5c).

It worth noting that, the 1st output at the first chart (figure 5.5a) converge correctly like 2nd output did. At the second chart (figure 5.5b) loss and precision of the 1st output were being too worse, because it is responsible for bigger "objects" than the 2nd output (figure 5.3). An extra reason to this fact is that there are not many big "objects" in the dataset, demanding more time to converge.



(a)



(b)

GPU Performance	Average Inference Time (ms)	FPS
Tesla K80	46.68	21.42
Tesla T4	29.26	34.18

(c)

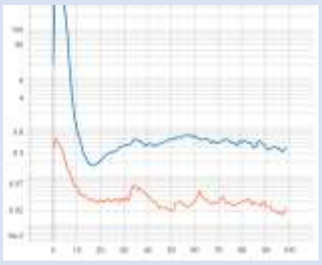
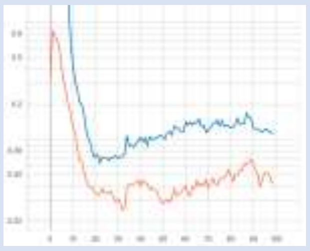
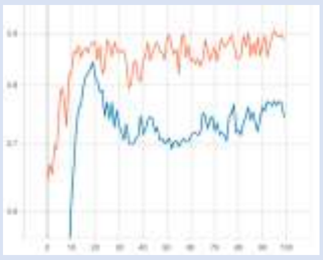
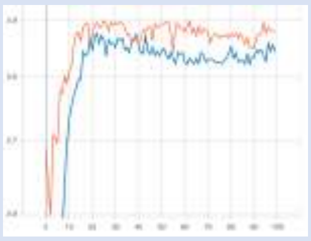
Figure 5.6: Outputs of the trained Network. . The predicted boxes have been drawn with blue color and score confidence of the class with red. The images (a) and (b) were neither in train-set nor in validation-set. They download from Google Search.

5.3. Compatibility Issues

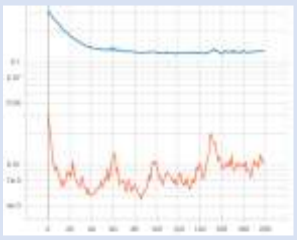
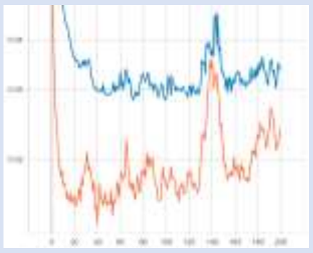
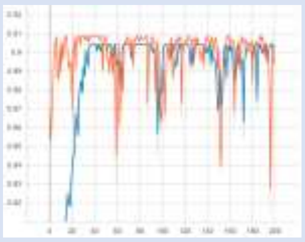
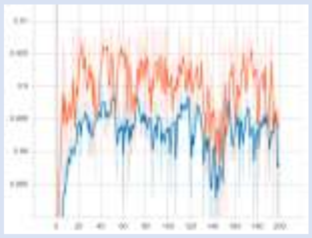
In order to convert the model being compatible and able to run on the Edge TPU, there were compatibility issues, as they have been mentioned in *section 4.2*. To convert the model being eligible, it needed to change some specifications of the Network like

- Replace the Leaky ReLU activation with ReLU,
- Replace Network's dynamic sizes with exact sizes to export a specific graph capable to run on the Edge TPU. (batch size definition to 1),

After making these changes there is the need of model's weights training again in order to make subtle adjustments. As we can see in figure 5.6, the effects of the changes are not so vital having high precision, but it is not so robust as before. The training steps followed the opposite direction as before. Firstly, the whole model's weights were trained, correcting Backbone's extracted features, and afterwards frozen it and converging the Detector's weights.

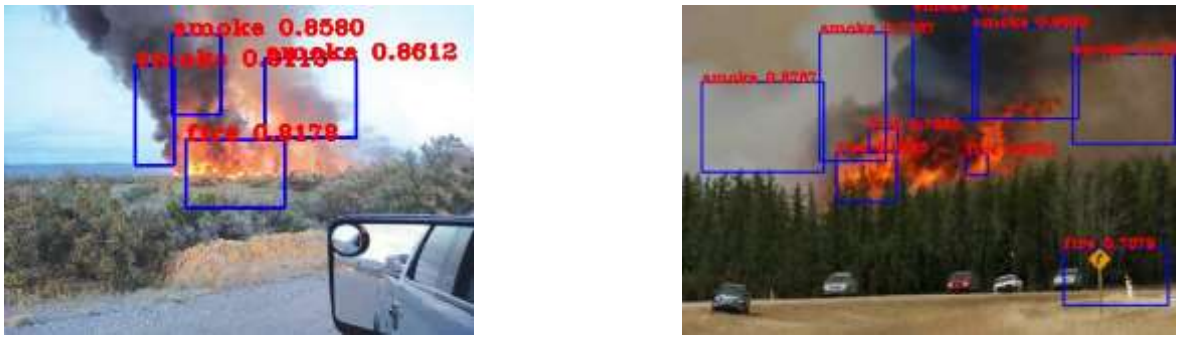
Stage-2	Output 0 (13x13)	Output 1 (26x26)
Loss		
Precision		

(a)

Stage-1	Output 0 (13x13)	Output 1 (26x26)
Loss		
Precision		

(b)

Figure 5.7: For each chart 1st row depict the loss update among the epochs and the 2nd row depict the precision ascent. The orange plot is the updated values from train-set, and the blue one is from validation-set. (a) Extracted images during the training of the whole network, (b) during the training of the Detectors.



(a) (b)

Figure 5.8: Outputs of trained network with compatible Edge TPU Layers. The predicted boxes have been drawn with blue color and score confidence of the class with red. The images (a) and (b) were neither in train-set nor in validation-set. They download from Google Search.

5.4. Weights in HDF5 binary format and Quantization

The next step is to create a file with the weights in order to be converted with the framework of the Tensorflow Lite.

Tensorflow has many backward compatibility issues among its framework versions. Having solved the model compatibility issues for which layers are compatible with Edge TPU and Coral Dev Board, then, the next problem was the model's file format and how to store the model weights in order to be ready for quantization conversion. This problem is solved by using the 2.1 version that it could be able to quantize a Tensorflow 2 model through a file.

For the integer post-training quantization does not require any modification to the network. However, this conversion process requires that you supply a representative dataset which uses the same original training dataset (the same data range). This representative dataset allows the quantization process to measure the dynamic range of weights and activations, which are critical to finding an accurate 8-bit representation of each weight and activation value. Moreover, it is prerequisite to specify that the converter must use uint-8 as inputs and outputs (fixed-point integers) and specify also that it must use int8 built-in ops (fixed-point integers).

5.5. Edge TPU Compiling

First and foremost (as described in the *section 4.3*) a model it needs to be eligible having a capacity of 8MB, approximately. For a dataset with 2 classes (fire, smoke). The total parameters are 6.298.480+ 2.375.424+2.304cl = 8.678.512 (as described in the *section 2.2*), where cl is equal to 2, and considering parameters type is float-32 (4Bytes), YoloV3-Tiny weights are going to be 34.7MB. After quantization (8-bit precision) the capacity of the file would be 8.27MB. The Edge TPU Compiler when gets a model that's internally quantized but still uses float inputs, the compiler leaves a quantize op at the beginning of your graph (which runs on the CPU). Likewise, the output is dequantized at the end.

After training and integer post-training conversion the Tensorflow Lite model it has come up with the final step, compiling for the Edge TPU. If the model does not meet all the requirements for compatibility issues is can still be compiled, but only a portion of the model will execute on the Edge TPU. At the first point in the graph where an unsupported operation occurs, the compiler divides the graph into two parts. The first graph that contains only supported operations that they have been compiled into an EDGE-TPU operation and they are able to run on it, and the rest of everything else is going to execute on the CPU as shown

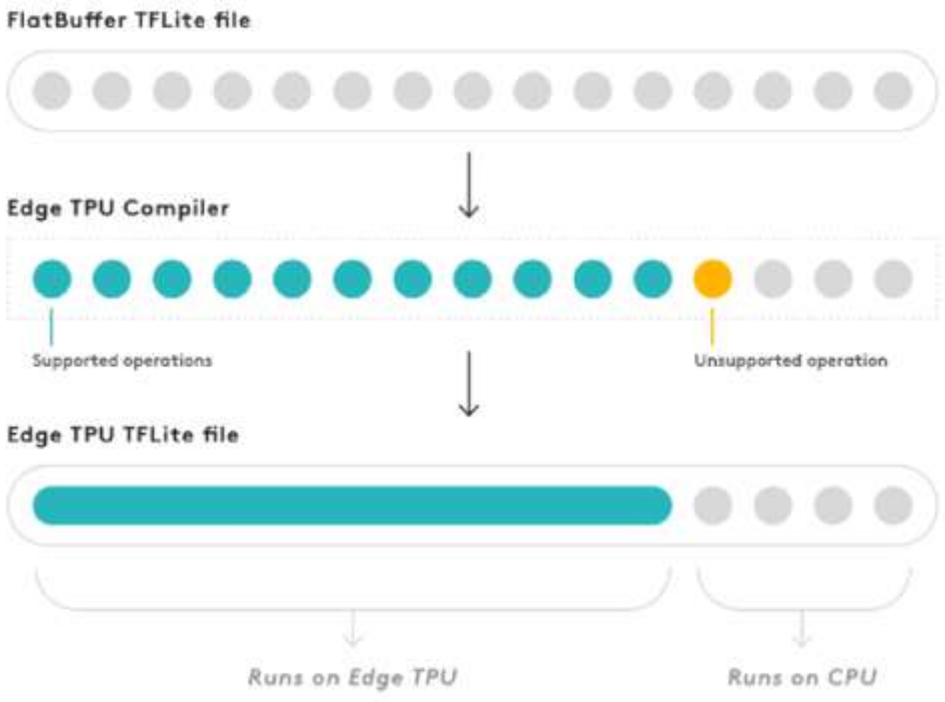


Figure 5.9: The compiler creates a single division for all Edge TPU compatible operations, until it encounters an unsupported operation. The rest runs on the CPU.

in figure 5.9. Moreover, considering two factors. First there are layers like Reshape that might not be mapped for large tensors, and as a consequence a part of the model would be uncached. Secondly, due to the fact that the model gets float inputs, then the output would be dequantized, and a message as shown below (figure 5.10) would be exported by the compiler.

```
Edge TPU Compiler version 2.1.302470888
Input: ./checkpoints_lite/yolov3_tiny_firedetect_relu.tflite
Output: ./edgetpu/tiny-firedetect/yolov3_tiny_firedetect_relu_
edgetpu.tflite

Operator                       Count      Status

MAX_POOL_2D                    6          Mapped to Edge TPU
RESIZE_NEAREST_NEIGHBOR        1          Mapped to Edge TPU
CONCATENATION                  1          Mapped to Edge TPU
QUANTIZE                       2          Mapped to Edge TPU
QUANTIZE                       2          Operation is otherwise
supported, but not mapped due to some unspecified limitation
CONV_2D                        13         Mapped to Edge TPU
RESHAPE                        2          Tensor has unsupported
rank (up to 3 innermost dimensions mapped)
```

Figure 5.10: The Edge TPU compiler export the message.

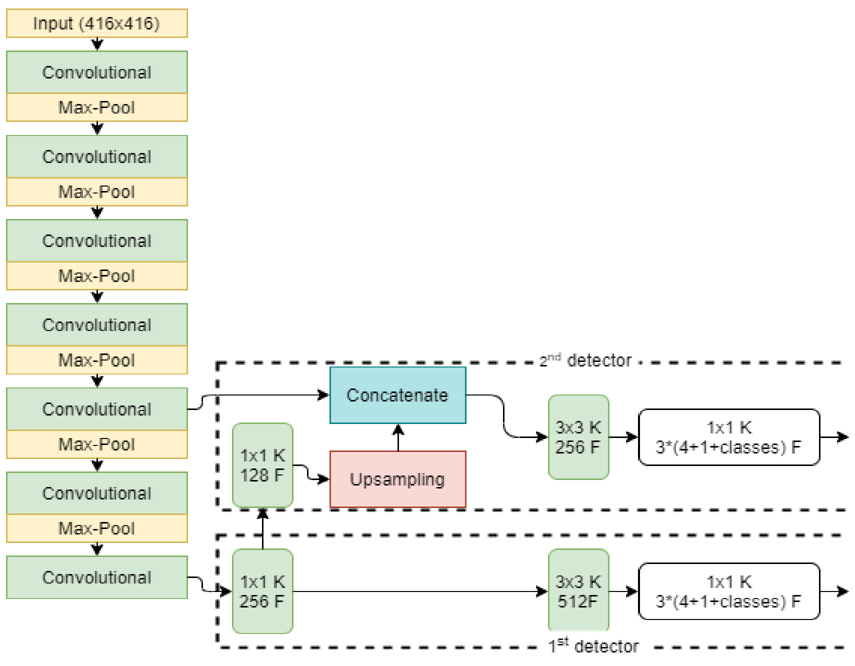


Figure 5.11: The Layers run on the Edge TPU according to the message. All the architecture except Non-Max Suppression.

5.6. Edge TPU Inference

To execute a Tensorflow Lite model it must run through an interpreter/ In the Python API, that is available with *Interpreter* class. In Tensorflow Lite each model is executed on the CPU, by default. This fails if your model was compiled for the Edge TPU, so it is necessary to instruct the interpreter to run the model on the Edge TPU. This is specified by delegates which are capable to handle graphs node through interpreter with the intention to run models, that are compiled for the Edge TPU. Essentially, using the Tensorflow API with the Edge TPU requires to instantiate the Interpreter by specifying the Edge TPU runtime as a delegate.

For example:

1. Add the delegate when constructing the interpreter.

```
import tflite_runtime.interpreter as tflite
EDGETPU_SHARED_LIB = {
  'Linux': 'libedgetpu.so.1',
  'Darwin': 'libedgetpu.1.dylib',
  'Windows': 'edgetpu.dll'
}[platform.system()]

tflite.Interpreter(
  model_path=model_file,
  experimental_delegates=[
    tflite.load_delegate(EDGETPU_SHARED_LIB,
      {'device': device[0]} if device else {})])
```

2. Now when a model runs, Tensorflow Lite delegates the compiled portions of the graph to the Edge TPU. Having specified the interpreter it needs to load the model to the memory which contains the model's execution graph. This is done in one line.

```
interpreter.allocate_tensors()
```

3. Considering that the Yolo architecture uses float inputs the next step is to transform the raw input to the quantized counterpart, using the equation $r = S(q - Z) \rightarrow q = \frac{r}{S} + Z$ (*section 3.4.2.*)

```
image = image / scale + zero_point
```

4. Now we are able to run the model by invoking the Interpreter, with the code line.

```
interpreter.invoke()
```

5. Next it needs to get the output and in turn to apply the sigmoid function getting the desirable parameters (object probability, bounding boxes, class probability) with the intention of passing them into the Non-Max Suppression.

```
output_details = interpreter.get_output_details()
pred_box, pred_obj, pred_class = yolo_boxes(output_details)
scores = pred_obj * pred_class
boxes, classes = non_max_suppression(pred_boxes, scores,
                      iou_threshold, score_threshold)
```

Due to the fact that Non-Max Suppression was not supported by Tensorflow Lite, it was attempted to written down a custom operation in C++ capable to run on the Edge TPU. But, due to the fact that there was not enough support, it was decided to write it by using numerical python library (numpy library).

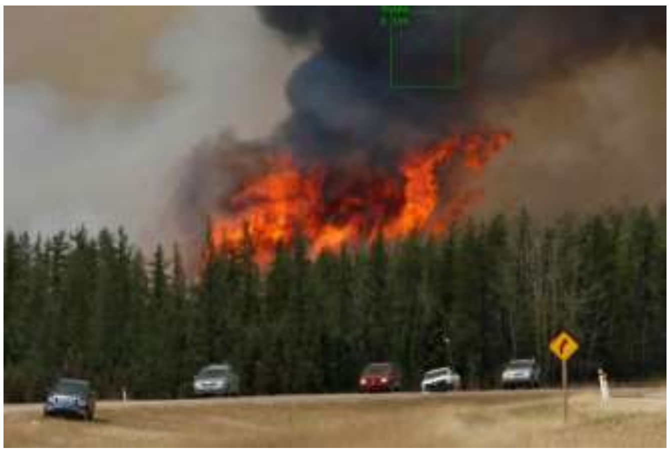
The Yolo architecture’s weights are trained and are based on the two vital factors for its operation, IOU-threshold, and Score-threshold. Due to the quantization and that the process of going from higher to lower precision is lossy in nature it has affected these two parameters. The method that is followed to tune them into the more reliable values was try and error. After, many efforts these IOU-threshold and Score-threshold went from 0.5 to 0.2 and from 0.5 to 0.15, respectively.

Finally, it is worth noting that the inference time of the YoloV3-Tiny architecture is 20.98ms for the Edge TPU and 109.53ms for the CPU. This is consequence of not be able to map the whole model into the Edge TPU, limiting its performance (figure 5.12).

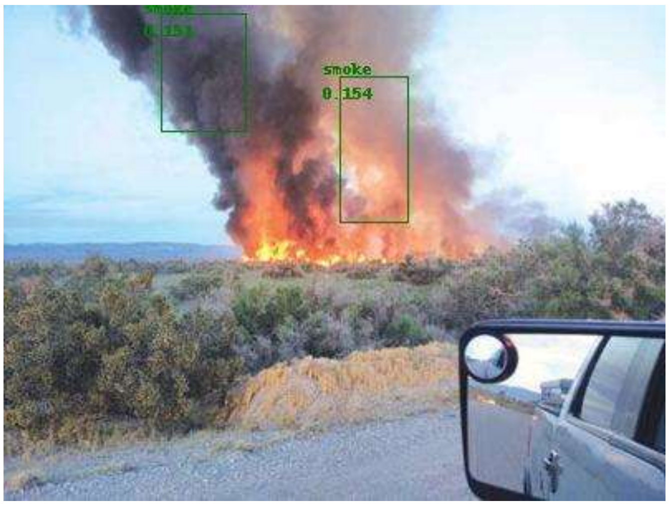
Performance	YoloV3-Tiny (ms)
Edge TPU	20.98
CPU	109.53
SUM	130.51

Figure 5.12: YoloV3-Tiny Performance while running on the Coral Dev Board.

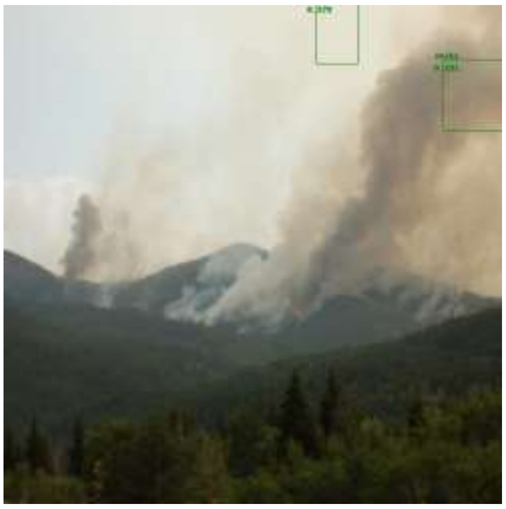
Some images extracted from Coral Dev Board using Edge TPU:



(a)



(b)



(c)

Figure 5.13: Outputs Detections from Coral Dev Board with Edge TPU

Chapter 6

Conclusion and Future Work

In this thesis, we propose a method for fire detection based on a reliable, frequent low-cost report of situation, using object detector and specifically the architecture You-Only-Look-Once (YOLO). Until recently, deep neural networks were not able to run on adaptable and portable machines and embedded systems, being power hungry and demanding a lot of computations. My approach was to valid whether an early fire detection could be implemented taking only visual stimulation by using the Google Coral Dev Board with Edge TPU in conjunction with deep neural networks. Having considered that the dataset is not that big as I would like to [13], but even with this the weights were able to converge with high precision. The results are extremely encouraging and positive. Evaluating the model in the Edge TPU with images that were not in the train-set or validation-set and making acceptable predictions may be encouraging.

Considering that, there are many factors that they can help to improve the final results. Some of them could be the compatibility among the Tensorflow versions for better development. Moreover, support for training quantized weights with the inception of being able to converge the weights during quantized adjustment and not by straight conversion. Better Edge TPU Compiler making it capable to map on the Edge TPU more operations without running on CPU (Non-Max Suppression). Additionally, it could be possible to change the architecture of the YoloV3-Tiny making it more robust, reliable with better performance in aspect of speed. It could be possible to replace the traditional Convolutional Layers with Depth-Wise Convolutional Layers which contain 9x less parameters (which means less capacity, which means better mapping on the Edge TPU) than traditional without affecting its performance. Finally, the introduction of infrared cameras are extremely necessary driving the project into the right direction of extra reliability and ensurance.

Bibliography

[1] Joseph Redmon, Ali Farhadi, “YOLOv3: An Incremental Improvement”, University of Washington

[2] Keiron O’Shea and Ryan Nash, “An Introduction to Convolutional Neural Networks”, Department of Computer Science, Aberystwyth University, Ceredigion, SY23 3DB, School of Computing and Communications, Lancaster University, Lancashire, LA1

[3] Sergey Ioffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, Google Inc., sioffe@google.com, szegedy@google.com, 2 Mar 2015

[4] https://github.com/rafaelpadilla/Object-Detection-Metrics

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, , “TensorFlow: A System for Large-Scale Machine Learning”, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16).

[6] Mart´ın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”, Google Research, November 9, 2015

[7] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, “TENSORFLOW EAGER: A MULTI-STAGE, PYTHON-EMBEDDED DSL FOR MACHINE LEARNING”, 27 Feb 2019

[8] Benoit Jacob Skirmantas Kligys Bo Chen Menglong Zhu, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”, Google Inc

[9] “NVIDIA TESLA GPU ACCELERATORS: The world’s fastest accelerators’, october 2014

[10] “NVIDIA TESLA P100 GPU ACCELERATOR: Infinite compute power for the modern data center”, october 2016

[11] “NVIDIA T4 TENSOR CORE GPU: GPU Acceleration Does Mainstream”, march 2019

[12] Diederik P. Kingma and Jimmy Lei Ba, “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”, University of Amsterdam, OpenAI, University of Toronto, 30 Jan 2017

[13] Mark Everingham, Luc Van Gool, and Christopher K. I.Williams, "The PASCAL Visual Object Classes (VOC) Challenge", M. Everingham University of Leeds, UK,

[14] Yosinski J, Clune J, Bengio Y, and Lipson H. How transferable are features in deep neural networks? In Advances in Neural Information Processing Systems 27 (NIPS '14), NIPS Foundation, 2014.

[15] Completed courses from https://www.coursera.org/, "Convolutional Neural Networks", "Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization", "Structuring Machine Learning Projects"

[16] Charu C. Aggarwal, "Neural Networks and Deep Learning", a textbook, IBM T. J. Watson Research Center International Business Machines Yorktown Heights, NY, USA.

[17] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, "You Only Look Once: Unified, Real-Time Object Detection", University of Washington, Allen Institute for AI, Facebook AI Research, 9 May 2016.

[18] http://pjreddie.com/yolo/

[19] https://github.com/tzutalin/labelImg

[20] https://www.gislounge.com/real-time-fire-mapping-and-satellite-data/

[21] https://www.nesdis.noaa.gov/content/imagery-and-data