



University of the Aegean
School of Engineering
Department of Financial and Management Engineering

CLASSIFICATION OF TRAFFIC SIGNS BASED ON OBJECT RECOGNITION

Kalliopi H. Vitlari

Supervisor: Prof. Ioannis Minis
Committee Members: Associate Prof. Nikolaos Ampazis
Assistant Prof. Vasileios Koutras

Chios, September 2021

To my Family

Acknowledgments

First and foremost, I am extremely thankful to my supervisor Professor Ioannis Minis who made this work possible and provided invaluable feedback on my analysis and framing, at times responding to emails late at night and early in the morning. His guidance and advice carried me through all the stage of writing my thesis.

Besides my advisor, I would like to express gratitude to Dr. Konstantinos Mamasis for his treasured support which was really influential in shaping my experiment methods and critiquing my results. Also, my gratitude to George Tepteris for the immense patience that showed and the helpful advices/hardware that he gave me.

Additionally, I would like to thank all the members of DeOPSys Lab of Department of Financial and Management Engineering and my colleague Cleo for the strong relationship and cooperation we have developed through this project.

My appreciation also goes out to my family and friends, especially my brother for always being there for me, for their encouragement and support of my decisions. I know they will never stop supporting me.

Abstract

Traffic lights recognition and classification play an important role in the realization of autonomous vehicles. This automated process uses video (frames) to recognize and classify traffic lights along the vehicle's path in real time. In this thesis, we adapt a proven deep learning model to recognize three classes (states) of traffic lights: green, red and yellow. The model is YOLOv3, it includes Darknet-53 and combines object detection and classification. The deep learning algorithms are implemented in Google Colab (a cloud platform developed by Google). The resulting Convolutional Neural Network (CNN) is trained using publicly available data sets that we modify to enhance the available training data.

Firstly, the training and validation datasets are generated. Secondly, ground truth bounding boxes -which define the class and the object in the images- are created and uploaded to the Colab environment that runs the algorithm for object detection. The algorithm preprocesses the images, creates bounding boxes that contain the object and adjusts the weights of some model layers. To obtain the most appropriate weights, we perform various training/validation experiments using combinations of available datasets. The experiments indicate that the addition of very clear photos, which contain only traffic lights in the training datasets, and in general photos in which traffic lights are part of a general environment, significantly help the training process.

We use the best performing weights to conduct a large case study that uses as input video footage taped from the streets of Thessaloniki, which contains numerous traffic lights in all three states. We divide the predictions into 3 categories: True, False and No predictions. Initially the study indicated a relatively low performance of the model caused by a high percentage of No predictions. To address this issue, we used more than one photographs of each traffic light-state combination and combined the related predictions. As a result, the percent of No predictions was reduced significantly, and the combined process yielded better results.

Περίληψη

Η αναγνώριση και η ταξινόμηση των σημάτων κυκλοφορίας παίζουν σημαντικό ρόλο στην υλοποίηση των αυτόνομων οχημάτων. Αυτή η αυτοματοποιημένη διαδικασία χρησιμοποιεί σαν είσοδό της βίντεο (συγκεκριμένα μοναδικές συνεχόμενες εικόνες που παράχθηκαν από το βίντεο) για να αναγνωρίζει και να ταξινομεί την κατάσταση των φωτεινών σηματοδοτών(φαναριών) κατά μήκος της διαδρομής σε πραγματικό χρόνο. Σε αυτή την διπλωματική εργασία, προσαρμόζουμε ένα αποδεδειγμένο μοντέλο βαθιάς μάθησης για να αναγνωρίσουμε τρεις κατηγορίες (καταστάσεις) φαναριών: πράσινο, κόκκινο και κίτρινο. Το μοντέλο είναι το YOLOv3 και περιλαμβάνει το δίκτυο Darknet-53. Συνδυάζει την αναγνώριση και την ταξινόμηση των αντικειμένων. Οι μέθοδοι βαθιάς μάθησης εφαρμόζονται στο Google Colab (μια πλατφόρμα που αναπτύχθηκε από την Google). Το νευρωνικό δίκτυο Convolutional Neural Network (CNN) εκπαιδεύεται χρησιμοποιώντας δημόσια διαθέσιμα σύνολα δεδομένων (public datasets) που τροποποιούμε για να βελτιώσουμε τα διαθέσιμα δεδομένα τα λεγόμενα training data.

Πρώτον, δημιουργούμε σύνολα δεδομένων εκπαίδευσης και επικύρωσης, γνωστά ως training and validation data. Δεύτερον, δημιουργούμε χειροκίνητα ένα βασικό πλαίσιο οριοθέτησης των αντικειμένων κάθε φωτογραφίας, γνωστό ως ground truth bounding box, για να προσδιορίσουμε την κλάση (κατηγορία) και το αντικείμενο της εικόνας και στη συνέχεια ανεβάζουμε τις εικόνες στον κώδικα (σε περιβάλλον Colab) για να γίνει η διαδικασία αναγνώρισης. Ο αλγόριθμος προ-επεξεργάζεται τις εικόνες, δημιουργώντας ειδικά πλαίσια οριοθέτησης (bounding boxes) που περιέχουν το αντικείμενο και προσαρμόζει τα βάρη ορισμένων επιπέδων του μοντέλου, γνωστά ως model layers. Για να αποκτήσουμε τα πιο κατάλληλα βάρη, εκτελούμε διάφορα πειράματα εκπαίδευσης και επικύρωσης (training and validation) χρησιμοποιώντας συνδυασμούς των διαθέσιμων συνόλων δεδομένων. Τα πειράματα δείχνουν ότι η εισαγωγή πολύ καθαρών φωτογραφιών που περιέχουν μόνο σηματοδότες (και όχι άλλα αντικείμενα) στα σύνολα δεδομένων της εκπαίδευσης με φωτογραφίες στις οποίες τα φανάρια αποτελούν μέρος ενός γενικού περιβάλλοντος, βοηθά σημαντικά τη διαδικασία εκπαίδευσης.

Χρησιμοποιούμε τα βάρη με τις καλύτερες επιδόσεις για να υλοποιήσουνε μια εκτεταμένη μελέτη περίπτωσης, η οποία χρησιμοποιεί ως είσοδο βίντεο που έχουν μαγνητοσκοπηθεί από τους δρόμους της Θεσσαλονίκης. Το βίντεο αυτό περιέχει μεγάλο πλήθος σηματοδοτών και στις τρεις καταστάσεις τους. Χωρίζουμε τις προβλέψεις σε 3 κατηγορίες: Σωστή πρόβλεψη, Λάθος πρόβλεψη και Ουδεμία (μη) πρόβλεψη. Αρχικά, η μελέτη έδειξε μια σχετικά χαμηλή απόδοση του μοντέλου που προκλήθηκε από υψηλό ποσοστό της κατηγορίας των μη προβλέψεων (No prediction). Για την αντιμετώπιση αυτού του ζητήματος, χρησιμοποιήσαμε περισσότερες από μία φωτογραφίες κάθε συνδυασμού φωτεινού σηματοδότη και της κατάστασής του και συνδυάσαμε τις σχετικές προβλέψεις. Ως αποτέλεσμα, το ποσοστό των μη προβλέψεων μειώθηκε σημαντικά και η απόδοση της ολοκληρωμένης διαδικασίας βελτιώθηκε ανάλογα.

Table of Contents

Chapter 1 Introduction	1
Chapter 2 Object detection methods and techniques	4
2.1 Image classification using Convolutional neural networks (CNNs)	4
2.2 Real Time Object Detection through the YOLOv3 algorithm	19
2.3 Current State-Of-The-Art	28
2.4 Contribution of this thesis	31
Chapter 3 Data preparation and training of the object detection model using the YoloV3 algorithm	32
3.1 Data collection, generation and labeling	32
3.2 Training and validation through Transfer Learning	37
3.3 Darknet-53+YOLOv3 model training	41
Chapter 4 Traffic light state detection: A case study	62
4.1 Experimental set up	62
4.2 Data processing	64
4.3 Results of the Neural Network Model	65
4.4 Improving the Results of the Neural Network Model	67
4.5 Conclusion	69
Chapter 5 Conclusions	70
References.....	73
Appendix A. Learning in MLP networks	79
Appendix B. Implementing Convolution Neural Networks in TensorFlow.....	110
Appendix C. Annotating images with ground truth bounding boxes	114

Table of Figures

Figure 2.1 VGG-16 Convolutional Neural Network (Smeda, 31/10/2019)	4
Figure 2.2 Kernel moves over the input to generate the output (Coursera(2020c)) ...	5
Figure 2.3 Representing a RGB image and applying convolutional W_0 Filter (Kernel) (Stanford Course)	6
Figure 2.4 Representing a RGB image and applying convolutional W_1 Filter (Kernel) (Stanford Course).	8
Figure 2.5 Max pooling (Coursera,2020c)	9
Figure 2.6 Results of code for Convolutions and max-pooling (Coursera(2020c)).....	12
Figure 2.7 Results of code for input (Coursera(2020c))	13
Figure 2.8 Rescaled image((Gandhi, 2021).....	14
Figure 2.9 Rotated Image (Elisha, 2020)	15
Figure 2.10 Shifted image ((Sarin, 2019)	15
Figure 2.11 Human Image (Coursera(2020c))	15
Figure 2.12 Human Image applied shear range (Coursera(2020c)).....	16
Figure 2.13 Girl image with zoom (Coursera(2020c))	17
Figure 2.14 Cat Image with Horizontal flip ((Balla, 2020))	17
Figure 2.15 Data augmentation consists of on-the-fly image batch manipulations. This is the most common form of data augmentation with Keras (Rosebrock, 2019)	18
Figure 2.16 Timeline of evolution of object detection algorithms (Zou, et al., 2019)	20
Figure 2.17 SSD architecture (Jiatu, 2018)	21
Figure 2.18 R-CNN model (Girshick, 2014)	22
Figure 2.19 Faster R-CNN (Ren, 2017).....	23
Figure 2.20 YOLOv3 architecture. (Vinh, 2020) (Jiatu, 2018)	24
Figure 2.21 Combination of the three techniques (Saxena, 2021).....	27
Figure 2.22 Testing on Bosch Small Traffic Lights dataset (Kozel & Robert, 2020) ...	29
Figure 2.23 Testing of Trained model on random image from Google (Kozel & Robert, 2020)	29
Figure 3.1 Traffic lights from Carla simulator	34
Figure 3.2 Traffic Lights of Bosch Small Traffic Lights Dataset	34
Figure 3.3 Images from SJTU Small Traffic Light Dataset.....	35
Figure 3.4 Images of Berkley DeepDrive Dataset	36
Figure 3.5 Series of convolutional layers model that are locked, and application of Transfer Learning method in the last fully connected layers (Coursera(2020c))	38
Figure 3.6 Transfer learning Steps (Leclerc, et al., 2018)	38
Figure 3.7 Detection Flow Diagram (Raza & Song, 2020)	39
Figure 3.8 Adding of new “trainable” layers (Almog, 2020)	41
Figure 3.9 Code for developing and training the traffic signal detection model	49
Figure 3.10 Results of Carla dataset (training loss and mAP).....	54
Figure 3.11 Results of training loss and mAP for Clear dataset	55
Figure 3.12 The training loss and mAP of Blurred dataset.....	56
Figure 3.13 Results of the combination of three datasets.....	57
Figure 3.14 Results of combination of CARLA and Clear datasets.....	58
Figure 3.15 Results of combination of CARLA and Blurred datasets	59
Figure 3.16 Results of combination of Clear and Blurred datasets	60
Figure 4.1 State of 1st and 2 nd from 180 traffic lights	63

Figure 4.2 Test data processing	65
Figure 4.3 False prediction results	66
Figure 4.4 States of 10th Traffic light	68
Figure A. 1 Neural network model of the example	80
Figure A. 2 Cost function (convex&non-convex)	82
Figure A. 3 The convex shape of a simple instance of the cost function	83
Figure A. 4 Convex between $h\theta(x)$ and Cost(J) if $y=1$	84
Figure B. 1 CNN with one Dense	111
Figure B. 2 A model used for multiple classification.	112
Figure B. 3 Data augmentation technique.....	113
Figure C. 1 Open Directory	115
Figure C. 2 The location of files.....	115
Figure C. 3 The save format (Pascal/VOC).....	116
Figure C. 4 The save format (YOLO)	116
Figure C. 5 Create\new RectBox button for drawing the ground truth bounding box ..	117

List of Tables

Table 2.1	Object detection algorithms comparison	28
Table 3.1	Colab and Colab Pro tools (Buomsoo, 2020).....	42
Table 3.2	Parameter modifications for training	44
Table 3.3	Upgrade names of datasets.....	53
Table 3.4	The combination of three datasets	56
Table 3.5	Combination of CARLA and Clear datasets	57
Table 3.6	Combination of CARLA and Blurred datasets.....	58
Table 3.7	Combination of Clear and Blurred Datasets.....	59
Table 3.8	Synopsis of the experiments	60
Table 4.1	Datasets used to obtain alternative weights	64
Table 4.2	Results of predictions.....	65
Table 4.3	Improvement method results	69

Chapter 1 Introduction

Deep learning is used to accelerate the solution of certain types of complex computational problems, such as in the fields of computer vision and natural language processing (NLP). In deep learning, the data scientist is not required to manually select the relevant features; instead, a deep learning model will learn the important features. In recent years, deep learning and Artificial Intelligence (AI) has been applied in various engineering fields including autonomous driving, for which it significantly accelerated research and eventually has moved it closer to reality.

An autonomous vehicle with complete self-driving capability (i.e., without a human intervention) must accurately -and in real time- comprehend traffic signs and traffic lights, avoid conflicts with other vehicles, humans, or obstacles, while remaining on the road to ensure safe and correct operation. To achieve this, various sensors (cameras, sonar, Lidar etc.) are used to provide the raw data to AI models that control the vehicle. The detection and identification of objects is also assisted by using multiple camera sensors since this achieves a better overall level of object detection accuracy in standalone driving systems (Choi, et al., 2019).

An object detection algorithm for autonomous vehicles should satisfy both a high detection object accuracy and real time detection speed. There are many state-of-the-art methods that use deep learning for image classification such as the Convolutional Neural Networks (CNN), as well as the ResNet and the DenseNet networks to name a few. In the past few years, many object detectors have been developed based on CNN (Wang, 2021). These detectors can be split into two categories: two-stage and single-stage. Two-stage methods (RCNN, Fast R-CNN, Faster R-CNN) are used to improve the detection speed. A region proposal¹ is generated in the first stage followed by the second in which object classification and bounding box regression are performed. In a

¹ The way a Region Proposal Network (RPN) works is that an image (of any size) is imported and the output is several rectangular object proposals, each with a unique objectness score.

single-stage method (SSD, YOLO, YOLOv2, YOLOv3 etc.) object classification and bounding box regression are performed concurrently -without including a region proposal stage- using a comprehensive feature extractor mechanism.

The scope of this thesis is to study, analyze, use and improve object detection methods that are appropriate for autonomous driving, focusing on the detection of traffic signal state (Red, Yellow and Green). Based on our study, the most appropriate method and model are selected, the model parameters are tuned, the model is trained and is tested using an original dataset encompassing numerous traffic lights of Thessaloniki. Based on the test results, appropriate refinements are made to improve model performance.

More specifically, the thesis

- presents and describes relevant aspects of the background in deep learning
- presents and describes the state-of-the-art in object classification and detection of traffic lights for autonomous vehicles
- uses Darknet-53 and YOLOv3 to classify and detect traffic lights images. Appropriate training is performed by executing multiple experiments to select and use the most appropriate training dataset
- presents and explains the new dataset that is created from the author for further testing
- tests the model with this dataset to evaluate the detection and accuracy performance of the model
- proposes a refinement stage to improve model performance.

For the necessary theoretical background in the aforementioned research areas, the author followed relevant Coursera courses² in:

- Machine learning (Stanford University)
- Computer vision basics (The State University of New York)
- Neural Networks and deep learning (DeepLearning.AI³)

² <https://www.coursera.org/>

³ An education technology company called DeepLearning.AI, develops a global community of Artificial Intelligence talent.

- Sequences, Time series and prediction (DeepLearning.AI)
- Introduction to TensorFlow for Artificial Intelligence, Machine learning and Deep learning (DeepLearning.AI)

The Google Colab tool was also utilized. This is an online environment which allows anyone to write and execute python code with zero configuration through a web browser to conduct relevant experiments (Li, 2020).

The structure of the remainder of this thesis is as follows: Chapter 2 provides an introduction to Neural Networks (NN), Convolutional Neural Networks (CNNs), models used for image classification and object detection, and discusses the architecture of networks and state-of-the-art methods applied for object detection in real time. Chapter 3 introduces the model adopted for traffic light recognition and the model's training process. It presents the datasets that are used and the steps of the process. Furthermore, it contains the results of the training experiments and the selection of the optimal dataset and network weights. In Chapter 4, the new dataset used for the test process is analyzed and the testing process and its results are presented. A new post-processing approach is proposed to improve model performance. The conclusion of the work and proposals for further research are included in Chapter 5.

Chapter 2 Object detection methods and techniques

In this Chapter we provide background information on the foundations of image classification, the Convolutional Neural Networks (CNN), and on a very effective model used in object detection, YOLOv3, which we used for the work of this thesis. For completion, a useful tutorial on the theory of Neural Networks is provided in Appendix A.1.

2.1 Image classification using Convolutional neural networks (CNNs)

The two main types of layers in a CNN are the convolutional layers and the pooling layers. For example, VGG-16, which is a CNN for classification, receives a picture as input and processes it through a set of convolutional layers, then through a pooling layer and this process continues until the fully connected layers and the SoftMax output layer (see Fig. 2.1).

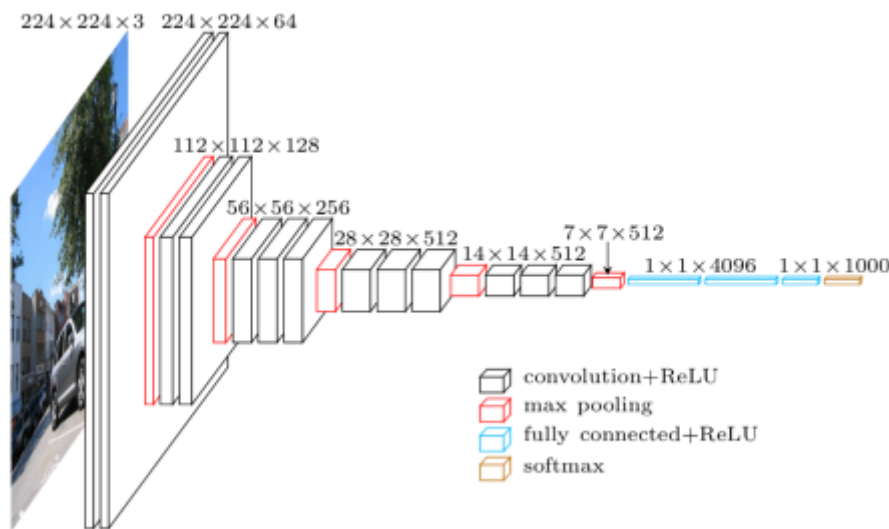


Figure 2.1 VGG-16 Convolutional Neural Network (Smeda, 31/10/2019)

2.1.1 Convolution filters and maxpooling operations

Consider an RGB image that is provided as input to the neural network. The image is an $M \times N \times 3$ array of pixels, where 3 corresponds to the three colors of the RGB

image (R=Red, G=Green, B=Blue). The width of the input picture is in the horizontal dimension, the height is its vertical dimension and the depth is the number of channels, that is three as stated above.

Zero elements are added around the image, an operation called padding. This operation results in a single pixel border added to the image with a pixel value of zero. Also, to assist the kernel with processing the image, padding allows for more space for the kernel to cover the whole image and leads to a more accurate analysis of images.

Convolution is the operation that modifies the above input when it is passed through a filter; in this case the filter is the convolution kernel. In CNN, multiple kernels are used to scan the input image. Each filter/kernel, slides from left to right across the image and continues this operation in each pixel row from top to bottom. The resulting output image is called feature map or activation map. 2D convolutions are usually used for black and white images, while 3D convolutions are used for colored images.

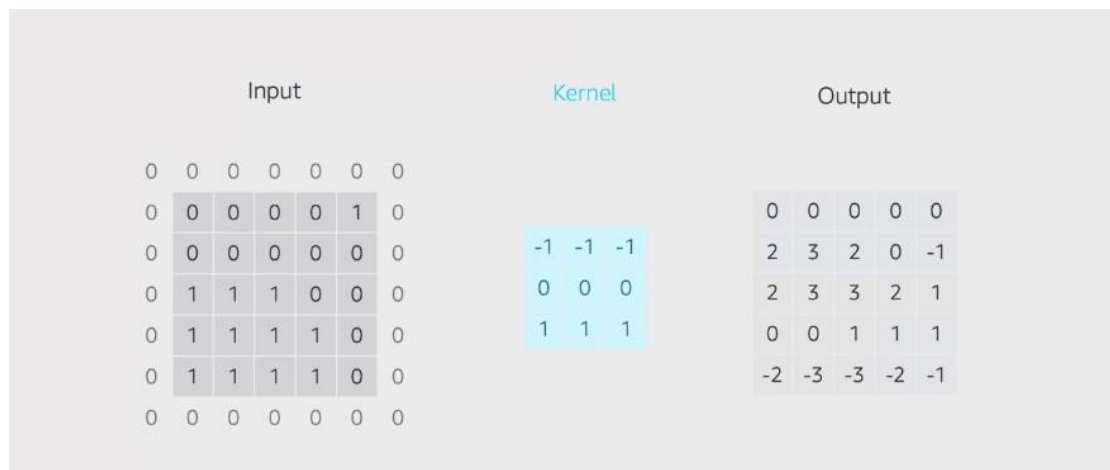


Figure 2.2 Kernel moves over the input to generate the output (Coursera(2020c))

There are many sizes of kernels, which can be used; for example of small (3×3) or larger kernel sizes (5×5). The most popular choice used by deep learning practitioners is (3×3).

The mathematical formula for the convolution operation in 2D is given by following equation. In this equation, the image is represented by matrix I , the kernel is K and i, j are the pixel indices on which the convolution is applied, also m and n are the width and the height of the kernel (Goodfellow, et al., 2016)

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n) \quad (2.1)$$

An explanatory example regarding the implementation and functionality of filters is given in Fig. 2.3.

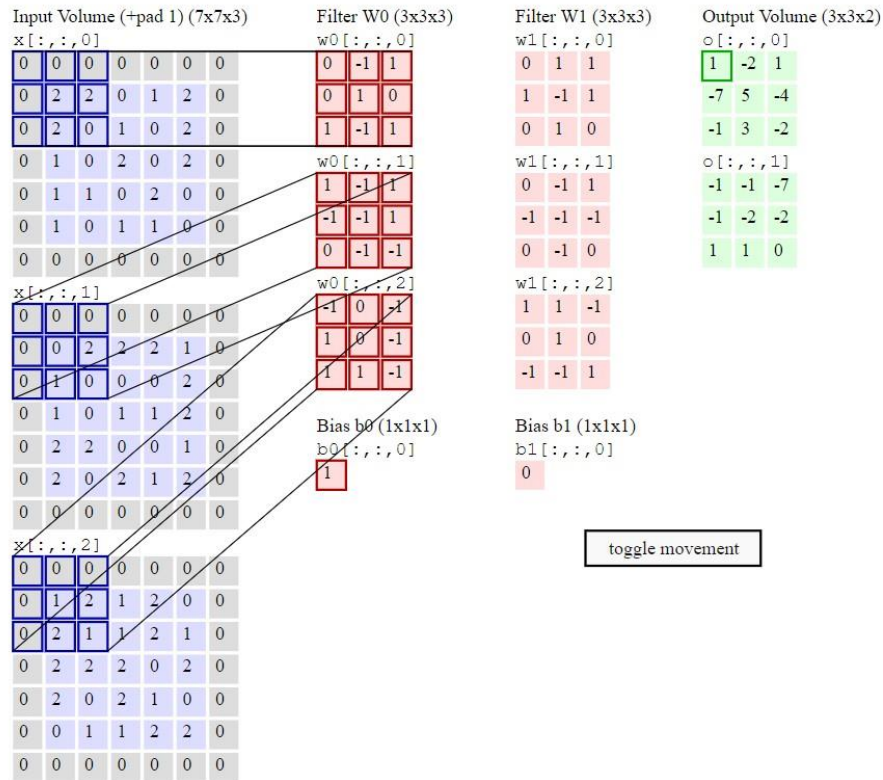


Figure 2.3 Representing a RGB image and applying convolutional W_0 Filter (Kernel)
(Stanford Course)

The first, second and third channels represent the red, green and blue colors respectively. In this case two kernels (filters) are applied, the first filter is the W_0 and second filter is the W_1 . The first green matrix contains the results of applying filter W_0

and the three channels. The second green matrix contains the output results from the second filter W_1 . Filter W_0 contains 3 different matrices, one for each channel.

Let's start with the first channel (7 x 7) input image with zero padding and we use the first (3 x 3) convolution filter to get an output image. The first step is to multiply the highlighted box in the input image with the first kernel. Each element is multiplied with an element in the corresponding location. Then, all the results are summed up, providing one value of the output. This is performed for each channel. The bias is 1 and is applied to the sum of the results of the three operations.

The first operation that involves the first channel and the corresponding kernel is:

$$\begin{aligned} (0 \times 0) + (0 \times (-1)) + (0 \times 1) + (0 \times 0) + (2 \times 1) + (2 \times 0) \\ + (0 \times 1) + (2 \times (-1)) + (0 \times 1) = 2 - 2 = 0 \end{aligned} \quad (2.2)$$

The same process is applied to the second channel:

$$\begin{aligned} (0 \times 1) + (0 \times (-1)) + (0 \times 1) + (0 \times (-1)) + (0 \times (-1)) \\ + (2 \times 1) + (0 \times 0) + (1 \times (-1)) + (0 \times (-1)) \\ = 2 - 1 = 1 \end{aligned} \quad (2.3)$$

For the third channel the first operation with its kernel is as follows:

$$\begin{aligned} (0 \times (-1)) + (0 \times 0) + (0 \times (-1)) + (0 \times 1) + (1 \times 0) \\ + (2 \times (-1)) + (0 \times 1) + (2 \times 1) + (1 \times (-1)) \\ = -2 + 2 - 1 = -1 \end{aligned} \quad (2.4)$$

Adding the results of the three operations and the bias (1), the output result is $0 + 1 - 1 + 1 = 0 + 1 = 1$ for the first output filter. This is shown as element (1,1) of the first green matrix in Fig. 2.7. This operation is repeated by moving the kernel to the right to get element (1,2) of the output. The step size of the kernel sliding across the image is called a stride. Here, the stride is 2. A stride size greater than 1 will always downsize the image. So, in order to find the other results for the first output, the highlighted box moves by a stride of 2 pixels horizontally, vertically, and horizontally again, and so on (for each color). In this case the output is a 3x3 matrix.

The same procedure will be followed by the second filter (W_1) as shown in Fig. 2.4. The second green matrix contains the results from multiplying the matrices from filter

W_1 and the three channels. The first green matrix contains the outputs results of applying filter W_0 . Here, the bias is 0.

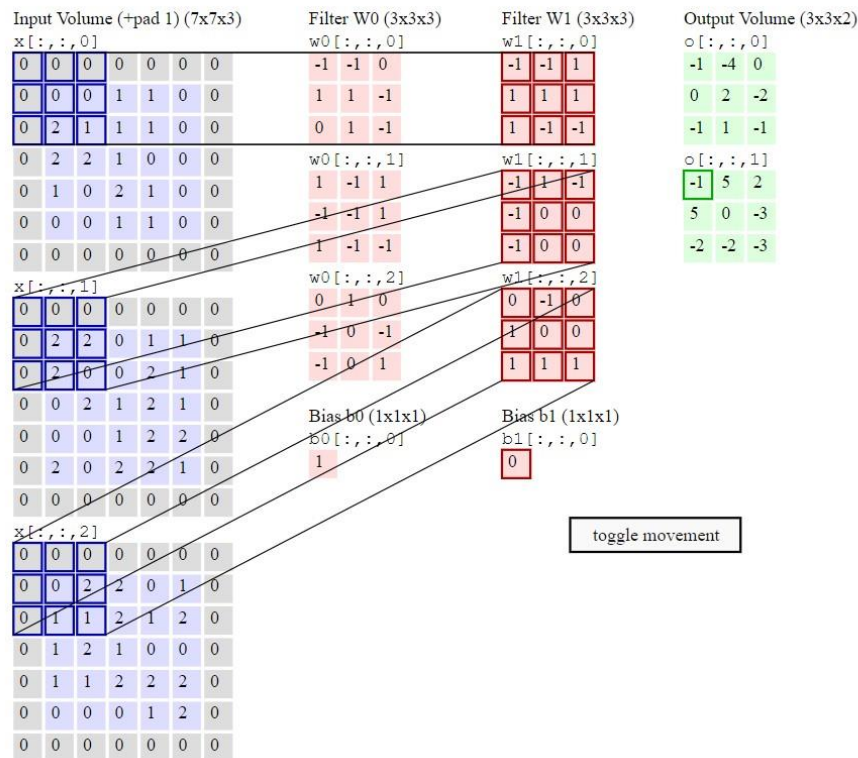


Figure 2.4 Representing a RGB image and applying convolutional W_1 Filter (Kernel) (Stanford Course).

As we can see in Figure 2.5, VGG -16 has 16 layers among quits multiple convolutional layers in particular are:

1. Convolution using 64 filters
2. Convolution using 64 filters + Max pooling
3. Convolution using 128 filters
4. Convolution using 128 filters + Max pooling
5. Convolution using 256 filters
6. Convolution using 256 filters
7. Convolution using 256 filters + Max pooling
8. Convolution using 512 filters
9. Convolution using 512 filters
10. Convolution using 512 filters+Max pooling

11. Convolution using 512 filters
12. Convolution using 512 filters
13. Convolution using 512 filters+Max pooling
14. Fully connected with 4096 nodes
15. Fully connected with 4096 nodes
16. Output layer with Softmax activation with 1000 nodes.

each convolutional layer including a large number of kernels; e.g. 64 filters (3×3) are applied in the first convolutional layer, 128 filters in the second one etc.

Convolutional networks may include pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of several neurons of one layer into a single neuron in the next layer. One of the possible aggregations we can make is to take the maximum value of the pixels in the group (this is known as Max Pooling). Another common aggregation is taking the average of the pixels in the group (Average Pooling). Max pooling is used to reduce the image size. In the case of Figure 2.9 if a 2×2 max filter is used and a stride of two, the output will be a 2×2 array.

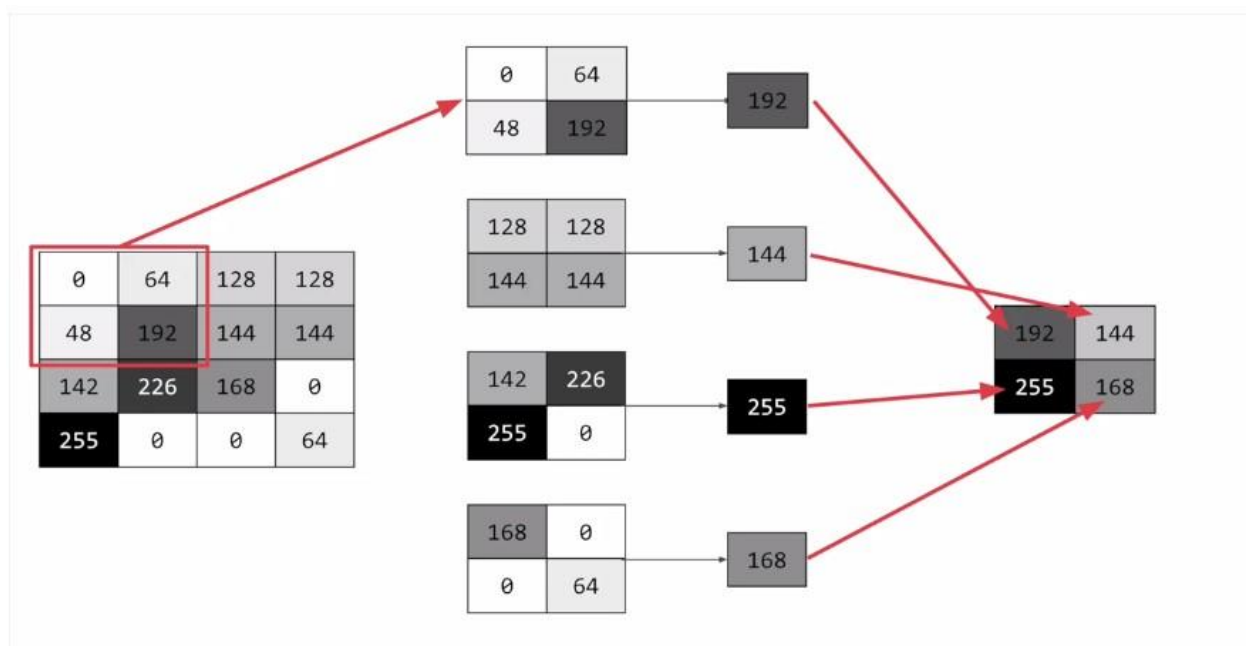


Figure 2.5 Max pooling (Coursera,2020c)

For every consecutive 2×2 window, only the maximum number is considered, as it can be seen in the middle part of the above Figure 2.9. Here, a filter of size 2×2 and a stride of 2 is applied. These are the hyperparameters for the pooling layer. In the first 2×2 window of the image with values (0, 64, 48, 192), the maximum is 192. So, the first element of the output takes the value 192. The same process continues for the next three outputs.

Consider as an example the VGG -16 network that consists of 16 convolutional layers, 5 pooling layers and 3 fully connected layers. The input is a $224 \times 224 \times 3$ array of an RGB image. The pre-processing layer takes this RGB image comprised of pixel values and subtracts the mean image value computed over the entire image.

- The first two layers are convolutional layers with 64 filters each; each filter has a 3×3 dimension (see Figure 2.5). The 3×3 filters have a stride of 1. In these layers, 64 filters are used that result in dimensions $224 \times 224 \times 64$.
- Next is the pooling layer with maxpool of 2×2 size and stride 2, which reduces the image size from $224 \times 224 \times 64$ to $112 \times 112 \times 64$.
- This is followed by two more convolutional layers, each with 128 filters, which results in the new dimension of $112 \times 112 \times 128$.
- Then maxpooling is used followed by
- another three convolutional layers are added with 256 each filters, which changes the size to $56 \times 56 \times 256$.
- Then a max-pool layer reduces the size further, followed by
- Three convolutional layers with 512 filters resulting in $28 \times 28 \times 512$.
- Finally, after max pooling and three last convolutional layers include 512 filters and result to a size of $14 \times 14 \times 152$
- This is succeeded by a max-pool layer with $7 \times 7 \times 512$ volume
- The $7 \times 7 \times 512$ output is flattened into a Fully Connected (FC) layer, which is followed by a SoftMax operation. “The fully connected layers perform classification of the significant features contained in each bounding box of the image (for the bounding boxes see the sub-section on object recognition

below). Finally, for the final detection the Softmax output layer is used which is a vector with a single score per class. The highest score usually defines the class of the contents of each bounding box". (Teptoris , 2020)

The underlying idea behind VGG-16 is simplicity. The focus is on having convolutional layers with 3 X 3 kernels (and always using the same padding). The max pool layer is used after a group of convolution layer with a filter size of 2 and a stride of 2.

Generally, convolutional layers are strong feature extractors in which the convolutional filters are capable of finding or picking up characteristics of images. The VGG-16 architecture is in the top 5 in terms of accuracy. It is sufficient to building powerful models with correct training and high validation accuracy.

2.1.2 Coding a Convolutional Neural Network with Pooling in TensorFlow

Let's now see how we implement CNN in Tensorflow.

First, we need to import all the libraries,

➤ `import keras_preprocessing`

The Keras dataset preprocessing utilities, located in *tf.keras.preprocessing*, help to go from raw data to a *tf.data.Dataset* object that can be used to train a model (Aakash, et al., 2021). The *tf.data.Dataset* represents a sequence of elements, in which each element is composed of one or more elements (Brain, 2021).

Images present in the dataset are in a variety of shape and sizes. For a neural network to be trained on these images, they must be in a certain shape. For a greyscale image, color depth of 1 byte (pixel) is used . For the images in color, there is a color depth of 3 bytes as they are in RGB.

Subsequently we use *tf.keras.models.Sequential*. The term "*sequential*" means that model creation involves defining a Sequential class and adding layers to the model one by one in a linear manner, from input to output. The example below (Fig. 2.10) defines a Sequential model that accepts image inputs with size 150×150 .

Next, *tf.keras.layers.Conv2D* is used, which takes as input the image of size $150 \times 150 \times 3$ (RGB). The number of filters depends on the type and complexity of the image

data. In general, the more features someone wants to capture in an image the higher the number of filters required in CNN. For 2D convolution we utilize the VGG-16 architecture, which uses multiple 3×3 filters. Since the first column and row as well as the last column and row are populated by 0, the image size becomes $148 \times 148 \times 3$, when padding is removed. The image is processed through the filters using the method mentioned above. The related (weight) parameters are $1,792 = 3 \times 3 \times 3 \times 64 + 64$.

Then, max-pooling is applied on the output of the first *tf.keras.layers.Conv2D* and the result of the Max-Pooling layer is $74 \times 74 \times 64$. Having passed through all convolutional and max-pooling layers, the output of the last *tf.keras.layers.Conv2D* will be flattened, and then the flattened neurons will be connected with each and every neuron of the next layer of 512 neurons.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 74, 74, 64)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 128)	0

Figure 2.6 Results of code for Convolutions and max-pooling (Coursera(2020c))

These latter operations are shown in Fig. 2.6. Firstly, 6272 is the output shape resulting from the last max_pooling (see Fig 2.10), that is, $7 \times 7 \times 128 = 6272$. Dropout is a technique to prevent overfitting. Specifically, dropout refers to ignoring units (neurons) during the training phase which are chosen at random. The selected units are not considered during a forward or backward propagation. *Dense* inserts a neural network with 512 hidden units (neurons) to use in feedforward and backpropagation. The 3,211,776 parameters result from the Flatten Dimension multiplied by the number of Neurons and adding the bias; that is, $(6272) * \text{Dense Dimension } (512) + \text{One bias per hidden neuron } (512) = 3,211,776$. The

output has three classes, and the number of parameters is Input Dimension (512) * Output Dimension (3) + One bias per output neuron (3) = 1539.

flatten (Flatten)	(None, 6272)	0
dropout (Dropout)	(None, 6272)	0
dense (Dense)	(None, 512)	3211776
dense_1 (Dense)	(None, 3)	1539
=====		

Figure 2.7 Results of code for input (Coursera(2020c))

During training, a multi-class loss function will be used since this is a multi-class classification problem. Furthermore, a SoftMax activation function will be used, which has non-binary outputs (3 classes).

For training the NN, the RMSprop optimization algorithm is used that is similar to the gradient descent algorithm. The RMSprop automates the learning rate tuning by using a moving average of the squared gradient. The latter utilizes the magnitude of recent gradient descents in order to normalize the gradient (See Appendix B.2).

2.1.3 Methods to avoid Overfitting

The models discussed above are overly complex with too many parameters. If the training dataset is not rich enough, the model may be overfitted. A model that is overfitted is inaccurate. Overfitting of a model may be easily assessed by monitoring its performance on both the training dataset and on a holdout validation dataset. Specifically, in our case with a very large number of parameters, the model produces good results in training data but, if overfitted, it performs badly on the validation data set. The goal of a deep learning model is to generalize well from the training accuracy to validation accuracy. This is very important for the model to produce accurate predictions.

2.1.4 Simplifying the model

The first method dealing with overfitting is to simplify the model. We may reduce the complexity of a model by simply removing layers or reducing the number of filters. This technique may reduce overfitting and is similar to the Dropout technique. Unfortunately, there is no general rule on how much to remove or how limited our

neural network should be, and, thus, we should resort to tests until finding the correct number of filters and layers.

2.1.5 Image augmentation

Image augmentation is a strategy focused on generation of new images from already-available ones. And more specifically, it's a technique that helps us reproduce an image in another form or dimension. Image Augmentation is a very simple, but very powerful tool to help avoid overfitting.

To do so we may use an image generator, which gives the flexibility of generating more images by executing any of the following techniques (see Appendix B.3):

✓ Scale

The final image size can be larger or smaller than the original image. Some pixels from the original image may be trimmed like the image below. The image can be scaled outward or inward. While scaling outward, the final image size will be larger than the original. Most of the time a part of the image is cut, with size equal to the original image. Inward scaling reduces the image size.



Figure 2.8 Rescaled image(*Gandhi, 2021*)

✓ Rotation

Rotates an image randomly in the range of 0-180 degrees.

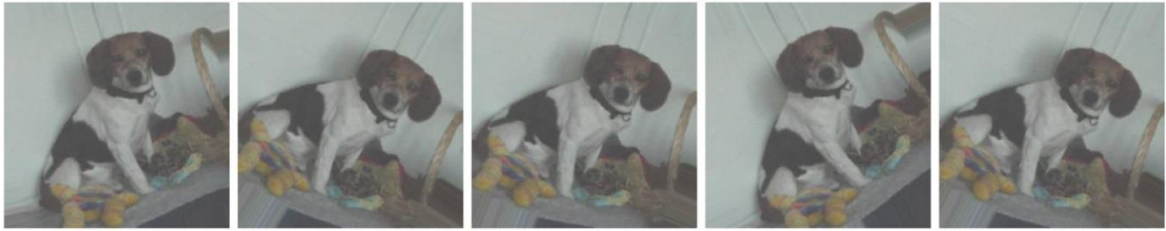


Figure 2.9 Rotated Image (*Elisha, 2020*)

- ✓ Width shift range, Height shift range

Shifting, moves the image around inside its frame. Many pictures have the subject centered. Training based on this kind of images might result in overfitting because they have a lot of features. The related parameters specify, as a proportion of the image size, how much the subject should randomly be moved around. For example, an image may be offset by 20 percent vertically or horizontally.



Figure 2.10 Shifted image (*Sarin, 2019*)

- ✓ Shear range

Consider the following image of Fig. 2.15.



Figure 2.11 Human Image (Coursera(2020c))

In this example, in the training set (left image of 2.12), there are no images of a person lying down. In the left image of Fig. 2.12 the human is standing up. In the right image (not part of the training data set) the person is lying down. To generate a similar image using the existing training set, one may shear the former image along the x-axis, its pose may end up very similar to the pose in the image on the right (see Fig. 2.13).



Figure 2.12 Human Image applied shear range (Coursera(2020c))

The shear parameter will shear the image by the specified amount. In the above example the shear is 0%.

✓ Zoom range

Zoom can also be very effective. For example, consider the following image on the right (not part of the training set). It is a woman facing to the right. If the training image (left image) is zoomed, it could end up with a very similar image to the one on the right.



Figure 2.13 Girl image with zoom (Coursera(2020c))

In this case, the zoom range will be a random value up to 20 percent of the size of the image. Depending on the size of the image we calculate the zoom values.

✓ Horizontal flip

Another useful tool is horizontal flipping. An image flip means reversing the rows or columns of pixels in the case of a vertical or horizontal flip respectively. To turn on random horizontal flipping, just write horizontal flip equals true in code and the images will be flipped at random. The following image shows a cat. In the left image, the right leg of the cat is lower than the other leg while the right image shows the opposite and thus horizontal flip is shown.



Figure 2.14 Cat Image with Horizontal flip (*Balla, 2020*)

✓ Fill mode

This fills in any pixels that might have been lost by previous operations. If the fill mode equals 'nearest' in the code, the pixel is filled using the value of its nearest neighbors to try and keep uniformity. Specifically, the closest pixel value is chosen and repeated for all empty values (see Appendix B.3).

For the implementation in TensorFlow, as always we need to import all libraries first.

➤ `from tensorflow.keras.preprocessing.image import ImageDataGenerator`

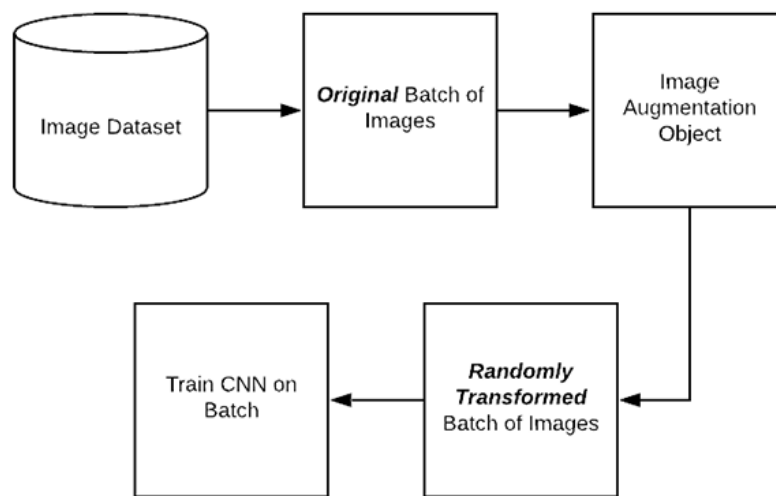


Figure 2.15 Data augmentation consists of on-the-fly image batch manipulations.

This is the most common form of data augmentation with Keras (Rosebrock, 2019)

In order to make the most of our few training examples, we will "augment" them via a number of random transformations, so that our model would never see twice the exact same picture. This helps prevent overfitting and helps the model generalize better.

In Keras this can be done via the `keras.preprocessing.image.ImageDataGenerator` class. This class allows us to:

- apply random transformations and normalization operations to our image data during training

- instantiate generators of augmented image batches (and their labels) via `.flow(data, labels)` or `.flow_from_directory(directory)`. These generators can then be used with the Keras model methods that accept data generators as inputs, `fit_generator`, `evaluate_generator` and `predict_generator`.

2.2 Real Time Object Detection through the YOLOv3 algorithm

In the previous section, the state-of-the-art methods for image recognition were presented. The current section deals with object detection, a subset of computer vision that detects and classifies the position of an object inside the image. Object detection algorithms have been extensively developed in recent years and the most widely used include Single Shot Detection (SSD), You Only Look Once (YOLO), Regional CNN (R-CNN) and the Faster R-CNN algorithms (Mantripragada, 2020). These algorithms classify the objects inside an image and specify the coordinates of bounding boxes around these objects, thus providing the exact location of the objects in respect to the bounds of the image.

Object detection methods: The state-of-the-art

Object detection and image classification are core computer vision (CV) problems with a distinct difference: Image classification aims to classify the image according to a set of pre-defined classes. Object detection, on the other hand, is more complicated: the aim is to classify the image into a class and also to detect the position of the object inside the image, using a bounding box (Ganesh, 2019).

In recent years, the use of faster hardware made deep learning implementations possible and gave rise to new methods (Figure 2.20) that solve the problem of object detection (Zou, et al., 2019).

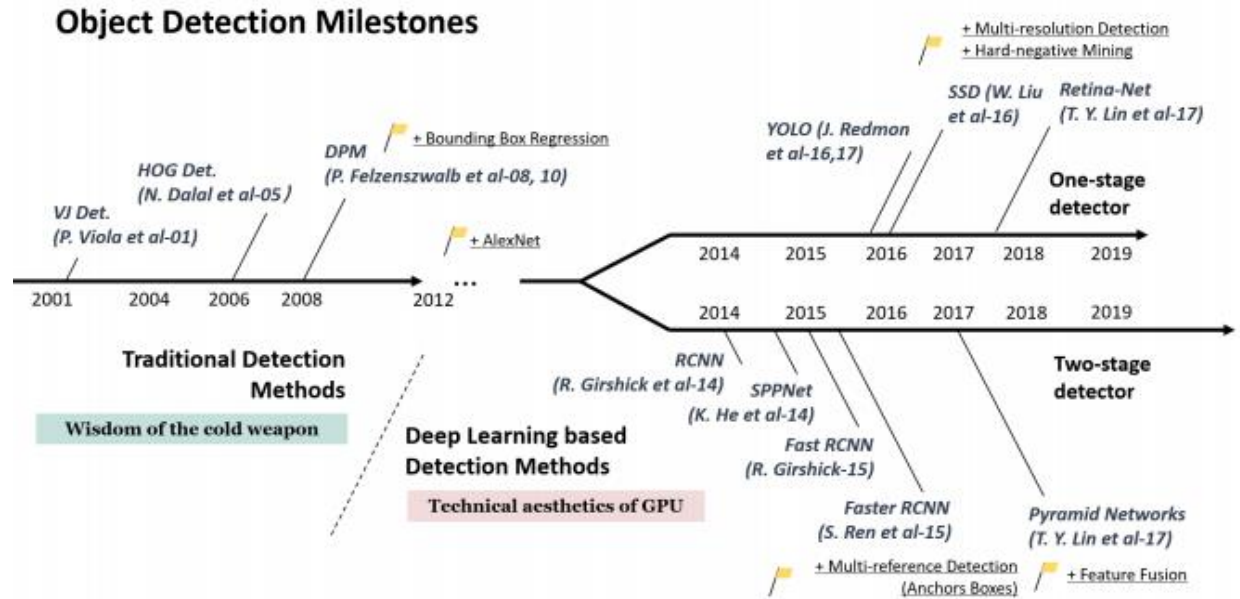


Figure 2.16 Timeline of evolution of object detection algorithms (Zou, et al., 2019)

2.2.1 Single Shot Detection

The Single Shot Detection (SSD) model (Wei Liu, 2016) was developed by Google and is based on a feed-forward Convolutional Neural Network (CNN) that extracts the features of the image into a “feature map”. The feature extraction is based on a small convolution kernel of size 3×3 that is applied to a series of convolutional layers. SSD predicts bounding boxes after multiple convolutional layers, with each layer focusing on different object size (small, medium, large) (see Fig. 2.17) (Liu, et al., 2015). Thus, in each layer, semantic meaning is extracted from the image by lowering the resolution of it. At the end of the convolutional steps, a classification probability is produced for each detected object and the coordinates of the bounding boxes around these objects are found. Finally, the SSD method applies a non-max suppression step (this technique keeps the one bounding box that fits the object perfectly) to produce the final detection results (Hosang, et al., 2017).

SSD simultaneously predicts the object bounding box and the object class as it processes the image. The basic steps are the following:

- The input image passes through a series of convolutional layers. The results are several sets of extracted feature maps at different sizes (Figure 2.17). SSD uses the Visual Geometry Group-16 (VGG-16) method to extract feature maps.

- A 3×3 kernel size convolutional layer is applied to each of these feature maps, to evaluate a small set of default bounding boxes. There are 4 types bounding boxes, each bounding box will have $(Number_of_Classes+4)$ outputs. Thus, Conv4_3 output has the size of $38 \times 38 \times (Number_of_Classes + 4)$, where 38×38 represents the size of the grid over the image, and 4 stands for the fact that for each grid cell there are 4 bounding boxes. If for example, there were 3 object classes, the output would be of size $38 \times 38 \times (3 + 4)$. In terms of number of bounding boxes, there are $38 \times 38 \times 4 = 5,776$ bounding boxes
- SSD predicts both the bounding boxes and the class probability simultaneously
- During training, the ground truth bounding box (a human drawn box that specifies the position of the object in the image and these predicted bounding boxes are matched, based on the Intersection over Union (IoU) method (see Section 2.3.4 “Intersection over Union”). The best predicted bounding box is the box that has an IoU -with the truth bounding box- larger than 0.5.

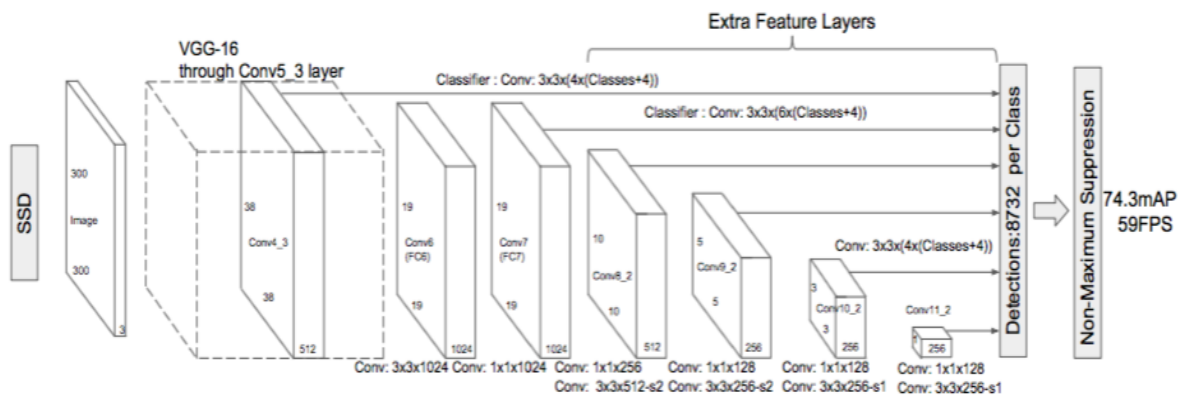


Figure 2.17 SSD architecture (Jiatu, 2018)

In addition to the Conv4_3 layer, which contains 5,776 bounding boxes and was discussed above, the number of the bounding boxes for the other convolution layers, after the Conv4_3 one, are listed below (Tsang, 2018):

- *Conv7* is $19 \times 19 \times 6 = 2,166$ bounding boxes (6 boxes for *Conv7*)
- *Conv8_2* is $10 \times 10 \times 6 = 600$ bounding boxes (6 boxes for *Conv8_2*)
- *Conv9_2* is $5 \times 5 \times 6 = 150$ bounding boxes (6 boxes for *Conv9_2*)

- $Conv10_2$ is $3 \times 3 \times 4 = 36$ bounding boxes (4 boxes for $Conv10_2$)
- $Conv11_2$ is $1 \times 1 \times 4 = 4$ bounding boxes (4 boxes for $Conv11_2$)

The total bounding boxes are $5.776 + 2.166 + 600 + 150 + 36 + 4 = 8.732$

2.2.2 R-CNN and Faster R-CNN

R-CNN (Figure 2.18) is an object recognition model (Girshick, 2014), which initially calculates the possible position of an object inside an image and then classifies the objects in the image. To find the position of an object inside an image, a selective search algorithm is used. The selective search algorithm, outputs approximately 2000 region proposals which are then fed to the CNN model to extract image features. The feature extraction method produces a 4.096-dimensional vector of image features and a Support Vector Machine (SVM) algorithm decides for the presence of an image class inside each region.

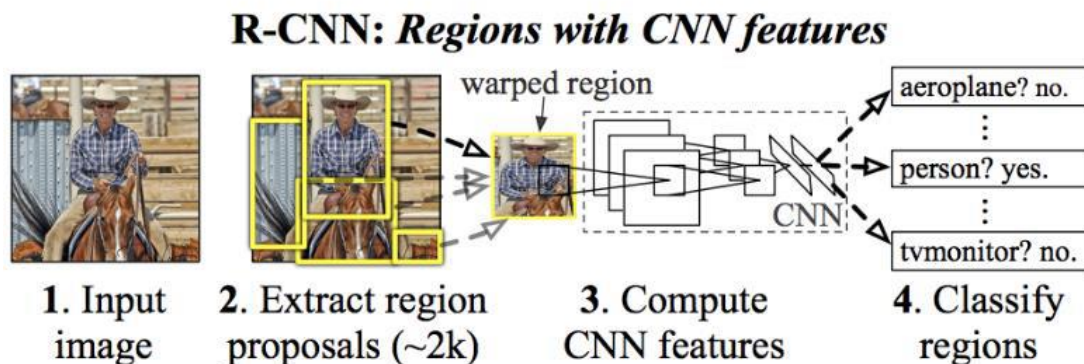


Figure 2.18 R-CNN model (Girshick, 2014)

The R-CNN model which is responsible for object recognition is inherently very slow. The 2.000 region proposals have a significant impact on the algorithm resolution time because the feature extractor must perform the same task for each one of these regions. Another problem is that, during the selective search, the network does not learn anything related with the patterns inside the image (Ren, 2017).

For these reasons, an improved model was developed (Ren, 2017) to address the inefficiencies of the R-CNN model. This improved model is called Faster R-CNN and does not use the Selective Search method for the region proposals; the model itself is

trained to predict region proposals using a CNN (Figure 2.19). These region proposals are then fed to a separate CNNs to decide if there is an object of interest inside this region. The output of the Faster R-CNN is both the class that the object belongs to and also its position inside the image.

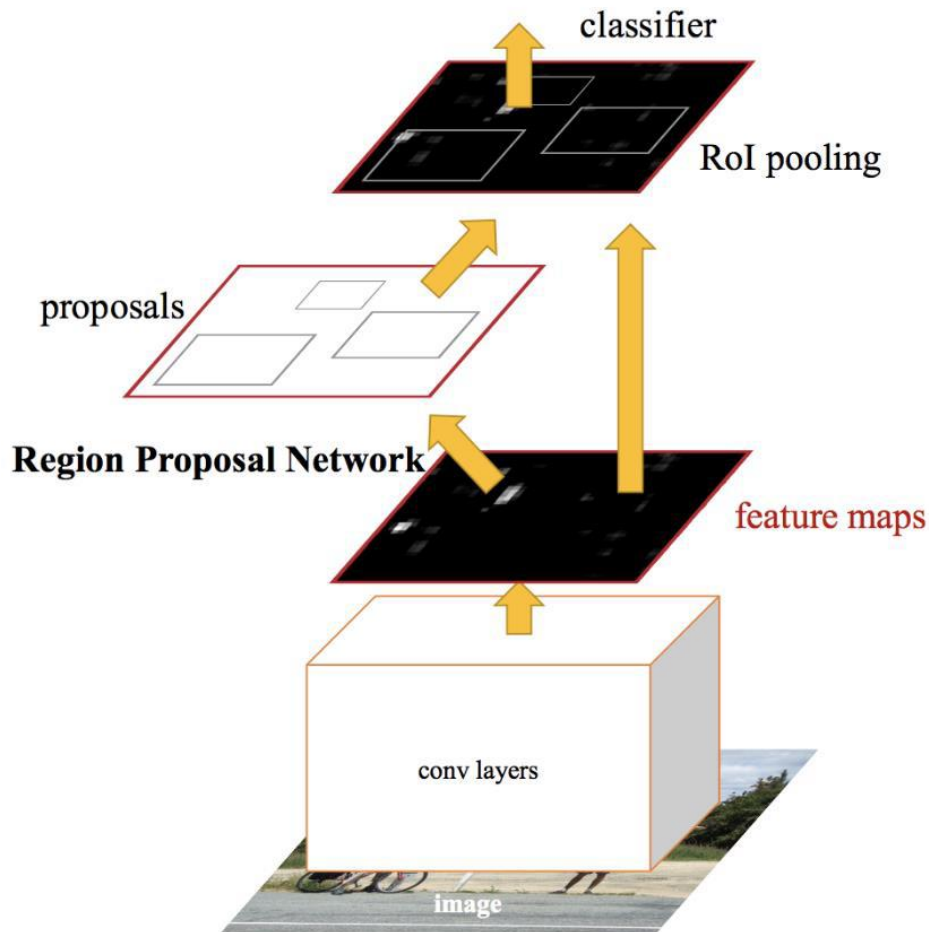


Figure 2.19 Faster R-CNN (Ren, 2017)

2.2.3 You Only Look Once (YOLO)

Overview

Most object recognition algorithms (such as the SSD approach discussed above) approach object detection as a classification problem. The YOLO architecture (Redmon, 2016) approaches the recognition part of the problem as a regression problem. A single Neural Network predicts the object's class and finds its location inside the image with just one "look". The YOLO's detection speed is about 10 times faster than other state of the art methods (Boesch , 2021).

The YOLO algorithm follows a simple approach: at first the image is resized into 448x448pixels. Then, the YOLO model divides the image into an $S \times S$ grid and assumes an object is centered in each grid cell. For each grid cell a bounding box and the probability of each class is predicted. The output is the class probabilities and the (t_x, t_y, t_h, t_w) coordinates of the object, which are provided if the class confidence is over a specific pre-set threshold.

The YOLOv3 model (Figure 2.20) contains 24 convolutional layers followed by two fully connected layers. These convolutional layers of the model are pre-trained on the ImageNet dataset; this makes the model easier to train in a transfer learning matter (Mwiti, 2021). The model uses the weights taken from the darknet-53 model (Redmon, 2016).

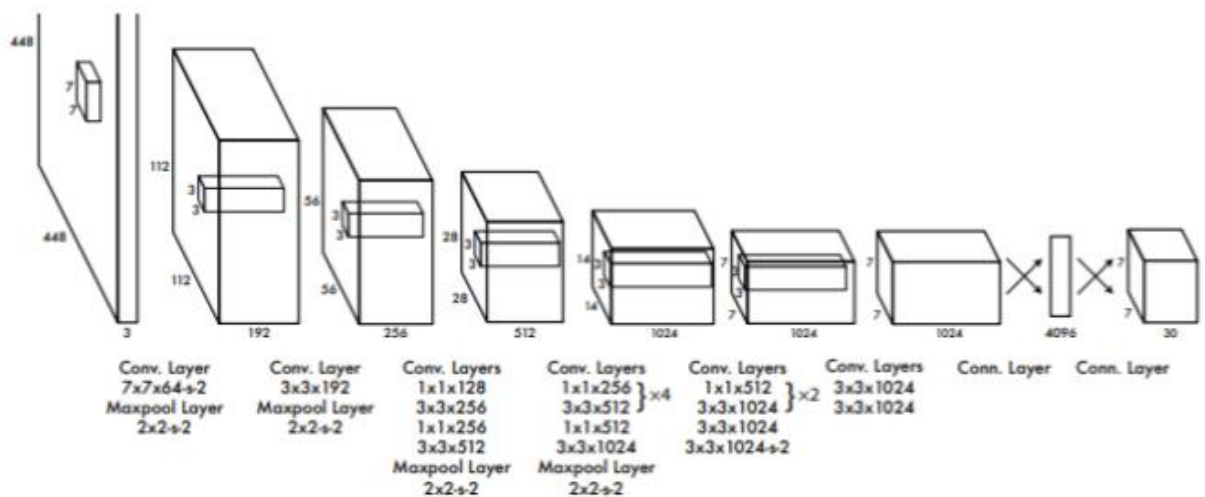


Figure 2.20 YOLOv3 architecture. (Vinh, 2020) (Jiatu, 2018)

Figure 2.24 shows the YOLOv3 network. YOLOv3 accepts 448pxx448px or 608pxx608px sized images, since this allows the processing of the images in batches which in turn speeds up the training of the network (Vignesh, 2020). For this reason, input images are resized to one of these fixed square dimensions. To extract the features such as color, shape and other aspects of the objects, multiple convolutions are applied to the image as it propagates through the network. The output layer is a

3D feature map and each depth channel represents a feature of the image or object (Vignesh, 2020).

The network characteristics are outlined below:

- The network has 24 convolutional layers for feature extraction and 2 fully connected layers for the output class scores and location coordinates (Fig. 2.24)
- The first 20 convolutional layers are followed by an average pooling layer and a fully connected layer is pre-trained on ImageNet
- The layers comprise 3×3 convolutional layers and 1×1 reduction layers.
- There are 64 filters that have a size of 7×7
- The output is a 7×7×30 vector which predicts the class probability and bounding boxes (Redmon, 2016).

Input and output of the YOLOv3 model

The input fed to the YOLOv3 model is a digital image (typically RGB) that includes an unknown number of objects.

The output of the model is an $S \times S \times (B \times (5 + C))$ tensor, where

- $S \times S$ is the size of the grid imposed by the model over the input image
- B is the number of bounding boxes for each cell. These boxes are positioned at the center of each object and have a different size and aspect ratio. Each bounding box is associated with five parameters: The box center coordinates (x, y) , the box height h , the box width w , and the probability P that the box contains an object
- C is the number of object classes. For each class, the output contains a conditional class probability value $P(Class_i|Object)$ which depend on the cell containing an object.

Thus, the output is a list of bounding boxes along with their coordinates, and the detected object class for each box. The 6 numbers $(P_x, b_x, b_y, b_h, b_w, C)$ associated with each bounding box are the following:

- b_x, b_y are the box's center coordinates
- b_w is the width of bounding box and b_h is the height of the bounding box
- P_x is the objectness score which represents the probability that an object falls within a bounding box
- C is the class that the object belongs to.

Initially, the image is divided into $S \times S$ grid cells with each grid cell containing 3 bounding boxes. These boxes are called "anchor boxes" and are used to predict the object present at the center of each grid cell (these boxes have different size and aspect ratio). Each cell can only predict one object for each box size (small, medium, large). This is done because many images can contain many objects of different sizes. In Fig. 2.21 there are three techniques of object detection. The first shows the grid cells on the input image. The second technique of the same Figure consists of the number of bounding boxes and the confidence score which explain the process of anchor boxes. The class probability map is derived simultaneously with bounding boxes + confidence, this process defines each object in a different color to identify the object and the class that it belongs.

The final step contains the outputs which include the object's position and the object's class. This technique is named non max suppression and is used to remove all bounding boxes, except the bounding box that fits the object perfectly (Hosang, et al., 2017).

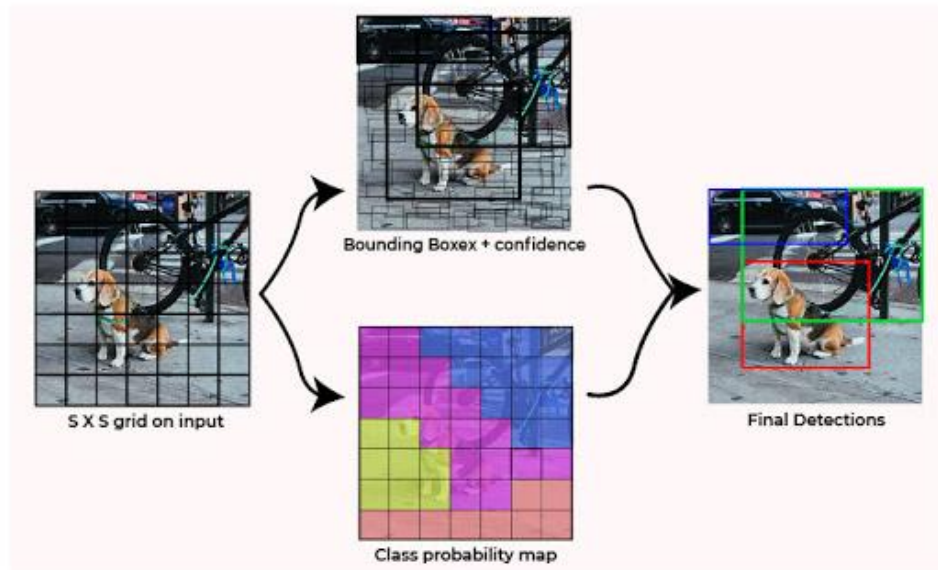


Figure 2.21 Combination of the three techniques (Saxena, 2021)

In Figure 2.21 there are lots of box predictions but only the best ones are kept in the final output. To select the best prediction box for each grid cell a non-max suppression method is applied to avoid selecting overlapping boxes. First, all boxes with a confidence score below a certain threshold are removed, then, the box with the highest probability is used to compute the IoU against all other boxes. If the resulting IoU is greater than another threshold (usually 0.6), the compared box is discarded (Sharma, 2018).

Training is based on the creation of the ground truth bounding boxes are created. These are hand labeled boxes that help to train the model in order to specify the position of the object in the image (Mohana, 2019). The ground truth bounding boxes are created using an annotation tool, described in more detail see Section 3.1. The IoU relates to the comparison between ground truth bounding box and predicted box.

2.2.4 Comparison between SSD, YOLO, R-CNN, and Faster R-CNN

According to Table 2.1 (Hui, 2018), YOLO achieves the best result in the best computational time (78.6% accuracy in only 91 Frames Per Second). Second is SSD, followed by Faster R-CNN and R-CNN. These results are expected since YOLO can detect and classify an object in one step. For this reason, real-time applications significantly favor the YOLO model (Hui, 2018). In this work we will use YOLO.

Table 2.1 Object detection algorithms comparison

Method	Mean Average Precision (mAP)	Frames Per Second (FPS)
SSD	74.3%	46
YOLO	78.6%	91
R-CNN	66%	2
Faster R-CNN	73.2%	7

2.3 Current State-Of-The-Art

Initially, the YOLO algorithm was slow (Redmon & Farhadi, 2018). Some necessary upgrades were made (changing of parameters for the choice of the better network will be used) and its successor, YOLOv3, became a fast detector of objects in photographs. YOLOv3 uses a pre-trained model called Darknet network and has higher accuracy than comparable models. In addition, the network structure utilizes the GPU more effectively, making it more efficient (Redmon & Farhadi, 2018).

Even though many object detection algorithms exist, public data sets of labelled pictures are often limited. Particularly for the problem in question, there are no datasets that contain labels of traffic lights. Two research papers that study traffic light detection and recognition are highlighted below.

2.3.1 Research Paper 1: Real-Time Traffic Lights Identification using YOLOv3 Algorithm for Autonomous Vehicles (Kozel & Robert, 2020).

Traffic lights and traffic sign detectors play a major role in autonomous vehicle safety. Although there are many methods that utilize a combination of a) image processing and b) training a neural network model, those methods are not fully accurate in detecting traffic light states. The root cause is the practical nature of the problem: a traffic light occupies just few image pixels in photographs taken from long distances and also, sunlight obstructs traffic light detection.

Kozel and Robert (2020) proposed to identify traffic lights and their three states in both urban and suburban areas with the development of a deep learning model which uses YOLOv3. The Bosch Small Traffic Lights benchmark was used for training. The Bosch

Small Traffic Lights dataset includes 5,093 pictures that contain thirteen classes organized in annotated files (see Fig. 2.22).



Figure 2.22 Testing on Bosch Small Traffic Lights dataset (*Kozel & Robert, 2020*)

The Mask Region-Based Convolutional Neural Network (Mask RCNN) method and the pre-trained weights from the COCO dataset were used. The training process lasted for 3 hours and for testing the trained network, generic traffic light images were downloaded from the internet (see Fig.2.27). Nevertheless, with the Mask RCNN model, the results were impossible and slow as prediction accuracy couldn't exceed 90% mainly because the vehicle lights were being detected as traffic lights.



Figure 2.23 Testing of Trained model on random image from Google (*Kozel & Robert, 2020*)

To improve the performance, YOLOv3 pre-trained weights from the Darknet network were used. Furthermore, the model was trained over 100 epochs. The trained model was able to detect traffic lights at a satisfactory detection rate even when vehicle lights were present in the picture (*Kozel & Robert, 2020*).

2.3.2 Paper: YOLOv3 Algorithm with additional convolutional neural network trained for traffic sign recognition (Novak, et al., 2020).

The most important ability of autonomous vehicles and most Advanced Driving Assistance Systems (ADAS) is the capacity to perceive all the static and dynamic objects around the vehicle (Novak, et al., 2020). Convolutional Neural Network (CNN) helps deliver safe ADAS in modern vehicles.

Based on the work by (Novak, et al., 2020) the YOLOv3 model has been pre-trained for the detection and classification of only five traffic sign objects. The YOLO algorithm is used to locate and detect objects in real time and then, an additional CNN is used to classify more specific subclasses of traffic signs. The CNN was trained in the code of YOLOv3 with excellent results on the test images. The dataset that was used for training is the Berkley Deep Drive Dataset, which contains limited examples of traffic signs for a total of 75 classes. For this reason, the additional CNNs that were used were trained on a different dataset which contains only traffic signs.

Two CNNs were created. The first treats traffic signs classes, while the second treats traffic signal classes. The difference between the two CNNs was only in the last Fully Connected layer. The training dataset contained 121,098 images, 95,020 of those were used for training, 23,773 for validation and 2,305 for testing. The first CNN - responsible for traffic sign type recognition- was trained for 20 epochs and the second CNN -responsible for traffic signal class recognition- was trained for 50 epochs. In conclusion, the accuracy of the algorithm was very high (close to 95%) for both categories. However, expanding The Berkley Deep Drive Dataset with new pictures took significant amount of time and effort even though it improved the predicted outcome (Novak, et al., 2020).

The results from the above 2 papers show that the YOLOv3 model has greater accuracy in object detection than other models. In addition, if some modifications are made to the CNN of the algorithm (for example the change of the epoch number) and if large datasets are used, then, the accuracy can reach 95%.

2.4 Contribution of this thesis

The objective of the thesis is to study and develop state-of-the-art techniques for traffic signal recognition, specifically traffic lights. This application is critical in driver-assistant systems and in autonomous vehicles. In order to accomplish these objectives we will perform the following steps:

1. Select the appropriate object recognition mode. As discussed above, there are several object recognition models, including SSD, R-CNN, Faster R-CNN, etc.). YOLOv3 is the preferred choice for our work for the reasons already discussed in Section 2.3.
2. Train the YOLOv3 Model, using appropriate, available datasets. Training involves a sequence of experiments in order to select the most effective datasets and training parameters
3. Acquire (develop) a new all inclusive data set of images, involving traffic lights under various conditions; i.e.
 - a. Traffic light state: red, yellow, green
 - b. Distances between the traffic light and the location from which the photo/video was taken.
 - c. Time and weather: day, night, rain, fog, sunshine.
4. Apply the trained model to the above original dataset. Improve its fidelity through new techniques
5. Draw conclusions and develop guidelines for training and implementation processes for this very important application.

Chapter 3 Data preparation and training of the object detection model using the YoloV3 algorithm

Prior to training the dataset to be used should be prepared carefully. The dataset should be in a special form, e.g. .xml or .txt. The .txt format is used in YOLOv3. Each image should be annotated with bounding boxes and hand labelled. To do so, an annotation tool is necessary. As a result, in the .txt file the information about the bounding box coordinates and the object classes are saved. There are multiple techniques used for this process in order to eliminate overfitting, normalize the data, augment (enrich) the dataset etc. This is discussed in Section 3.1.

The Section 3.2 explains in detail the steps that were followed to make the training process and the pre-trained weights that downloaded from the Darknet-53⁴ network. This technique is called transfer learning. Additionally, the YOLOv3 model is used and analyzed in detail in Section 3.3. Furthermore, the results of the custom training process, for training and validation datasets, were analyzed with Figures of the average loss and the mean average precision for object detection.

3.1 Data collection, generation and labeling

There are many publicly available open labelled datasets, including ImageNet (Yang, et al., 2021), Common Objects in Context (COCO) (Tsung-Yi, et al., 2015), Google's Open Images (Duerig & Krasin, 2016) etc. Each is a set of digital photographs with different states (Malevé, 2019) that developers use to train and validate the performance of their algorithms. The algorithms are said to learn from the examples contained in the dataset.

In this Thesis, four datasets have been used. The datasets contain (not exclusively) three classes, each for one state of a traffic light: Red, yellow, green. The Four datasets are:

⁴ [ImageNet Classification \(pjreddie.com\)](https://pjreddie.com/image-net/)

Table 3. 1 The datasets that are used for training

Dataset	Source	Number of images included on the datasets	Number of images used
Large Dataset	(Srivastana, 2017)	3,299	1,000
Bosch Small Traffic Lights Dataset	(Kaggle, 2020)	13,427	582
SJTU Small Traffic Light Dataset	(Xue, 2020)	5,786	1,217
Berkley DeepDrive Dataset	(Yu, 2021)	100,000	2,873

The “Large Dataset” (see Fig. 3.1) is downloaded from GitHub⁵. This dataset contains 3,299 images classified in 3 useful folders. The first contains 971 images with red traffic lights; the second contains 255 images with yellow traffic lights (see Fig. 3.1) and the third folder contains 145 images with green traffic lights. The size of each image is 224×224 (in pixels). All images are derived from the Carla autonomous car simulator program. “CARLA is an open source simulator for autonomous driving and has been produced by a team from the Computer Vision Centre at the Autonomous University of Barcelona, Intel and the Toyota Research Institute using the Unreal computer game engine.” (Teptoris , 2020). In these images the traffic lights are very close to the camera. Furthermore, the image contains only traffic lights and no other objects (such as road, signs, pedestrians, cars etc.) except general background. In our case we used only 1,000 images in model training, in order to maintain an appropriate balance with real photographs (that is, 369 from the folder with red traffic lights are not used).

⁵ <https://github.com/level5-engineers/system-integration/wiki/Traffic-Lights-Detection-and-Classification>

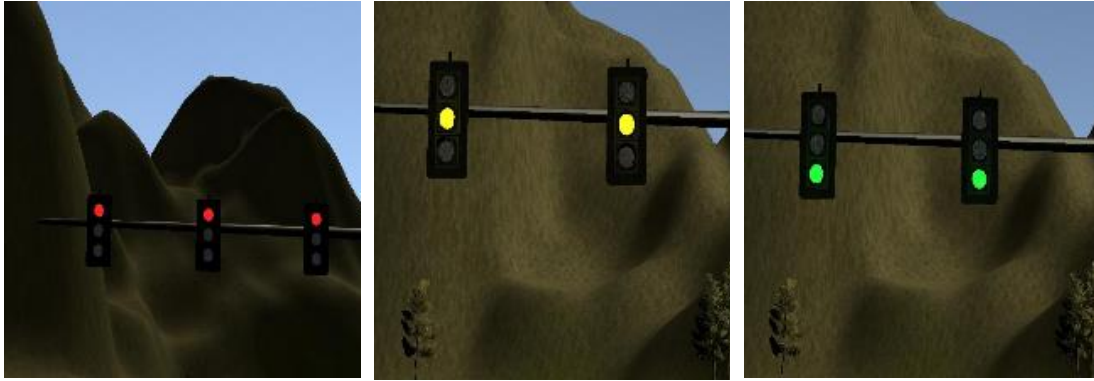


Figure 3.1 Traffic lights from Carla simulator

The second dataset is “Bosch Small Traffic Lights Dataset” from the Kaggle⁶ platform. Kaggle is an online platform that allows users to identify free datasets, explore and build models in a web-based data-science environment. This platform hosts data scientists, machine learning engineers and holds designs competitions to solve data science challenges (Lardinois, et al., 2017). The dataset contains 13,427 camera images. The dimension size of each image is 1280×720 pixels. All camera images are RGB (red, green, blue) and, in addition to traffic lights, include cars, road and multiple other objects. See examples in Fig. 3.2. Only 582 images are used to train the model; i.e. the ones that contain traffic lights at an appropriate environment



Figure 3.2 Traffic Lights of Bosch Small Traffic Lights Dataset

⁶ <https://www.kaggle.com/>

The third dataset, “SJTU Small Traffic Light Dataset” is downloaded from GitHub. It contains 5,786 images which are separated into two categories or “resolutions” a) of 1080×1920 pixels and b) 720×1280 pixels. It also contains 5 categories of traffic lights (red, yellow, green, off and wait on). Only the 3 categories were used (red, yellow and green) containing 1217 images, which are very clear. Figure 3.3 shows some sample images from this dataset.



Figure 3.3 Images from SJTU Small Traffic Light Dataset

The fourth dataset, “Berkley DeepDrive” (BDD100K) is downloaded from the Kaggle⁷ site. The package consists more than 100,000 HD videos recorded at various times of the day, seasons and weather. The data were collected from 4 locations (San Francisco, Berkeley, Bay Area and New York). The dimension of each image is 1280×720 pixels. From the classes of traffic lights 2,873 images are used for training the model (see Fig.3.4). From these images, 2,005 are the original ones and the other (868) have been created by data augmentation. The data augmentation techniques (see section 2.2) applied are the following:

- a) Horizontal flip

⁷ https://www.kaggle.com/solesensei/solesensei_bdd100k

- b) Vertical flip
- c) Shear range by $\pm 15^\circ$ horizontal and $\pm 15^\circ$ vertical

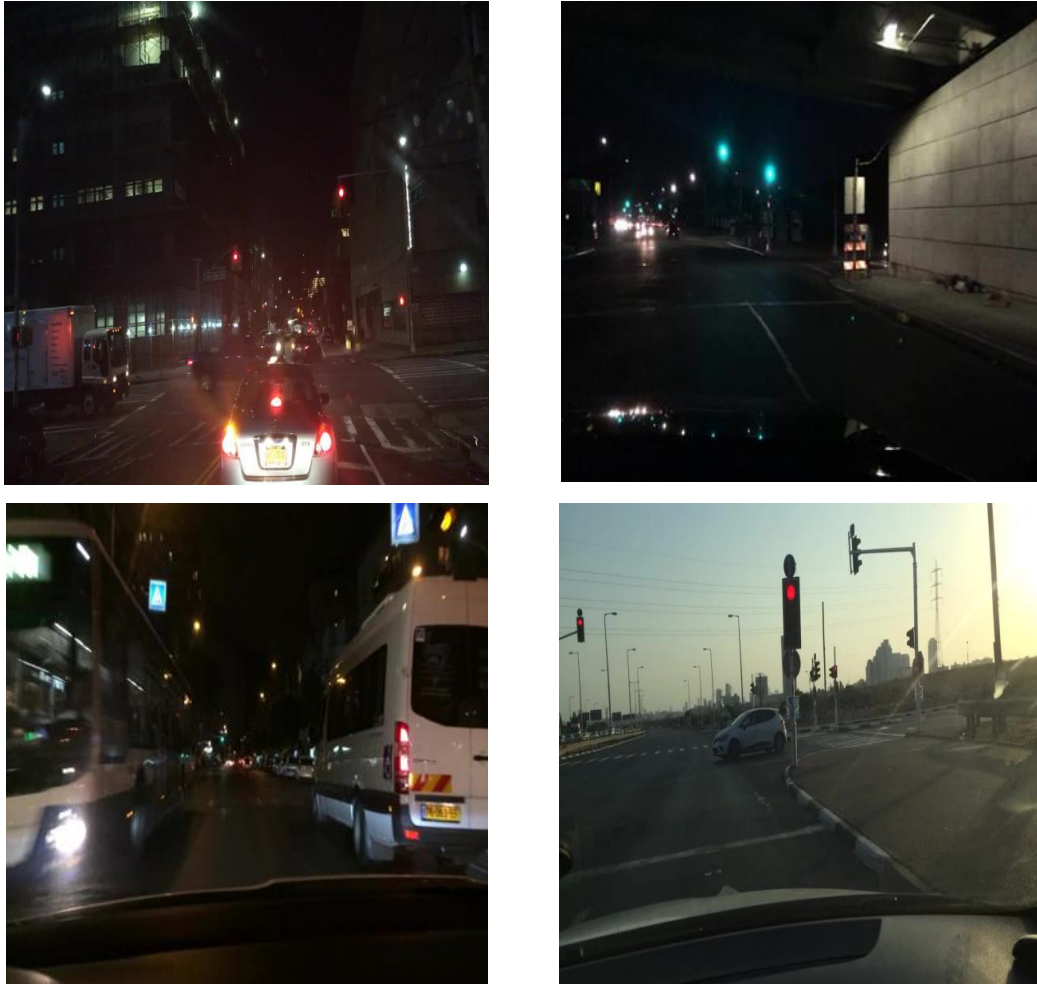


Figure 3.4 Images of Berkley DeepDrive Dataset

For training our network, the images in the above datasets were divided to three new sets. In each sets the images were allocated (randomly) into validation and training subsets.

- The first set contained the images taken from the Carla simulator (Large Dataset). There are 1000 images which are split into 700 images for training and 300 for validation (usually all images of datasets are divides into 70% for training and 30% for validation).
- The second set contained 1,217 images from the “SJTU Small Traffic Light Dataset”, 852 for training and 365 for validation. The images in this dataset are

clear (day or night) photos of actual scenes that contain multiple objects in addition to traffic lights.

- The third set contained images from the Berkley DeepDrive and Bosch Small Traffic Lights datasets. In total it contained 3,455 images, from which 2,419 were used for training and 1,036 for validation. The images in this dataset are not clear (blurred) photos of actual scenes, again containing multiple objects, in addition to traffic lights. These photos are also taken during the day or night.

All images in the above datasets do not include ground truth bounding boxes. However, these are necessary to perform the training of our Yolo3 network. Thus, in order to train our custom model, we inserted the ground truth bounding boxes in the traffic lights which are represented in all images. Do to so, we used the LabelImg⁸ graphical image annotation tool which is open source. The annotation tool and the manual process is described in Appendix C.

3.2 Training and validation through Transfer Learning

The concept of overfitting and its challenges were presented In Section 2.2.3. As already discussed, overfitting occurs when a NN with many parameters is trained using a limited dataset; this results in poor overall performance. To address this issue, very large data sets could be used to train the NN model. However, obtaining such datasets is a very hard and expensive process. Small datasets that are easier to obtain may be somehow enhanced with the methods presented in Section 2.3 but, although these methods help, oftentimes are not sufficient to address the overfitting problem.

This Section describes Transfer Learning which refers to the practice of using the weight parameters of a NN -that has been pre-trained on a large dataset- to classify real world images. Such a large dataset is ImageNet (Deng, et al., 2009), which is a 150GB dataset containing more than 1.2 million real-world labelled images organized in 1000 categories and it is one of the most widely used datasets in modern computer vision (CV) research.

Transfer learning capitalizes on the features that the model has already learned. Especially where only a small training dataset is available for a new NN model, the

⁸ Github repository of darrenl tzutalin: <https://github.com/tzutalin/labelImg>

weights of the pre-trained NN model help initialize the weights of the new NN model. In such cases, only the weights of the last few layers of the new NN model are adjusted through training. In this way, training addresses the overfitting issue.

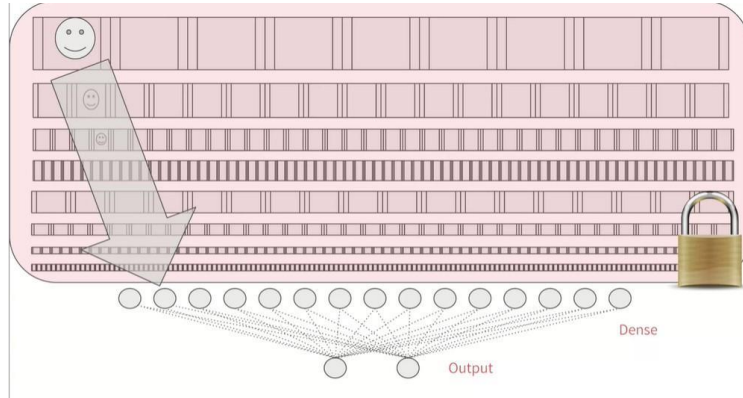


Figure 3.5 Series of convolutional layers model that are locked, and application of Transfer Learning method in the last fully connected layers (Coursera(2020c))

Figure 3.5 shows the concept of Transfer Learning: The pre-trained convolutional layers (shown in red in the figure) are locked and cannot be retrained using additional data obtained from a new dataset. These locked layers have already extracted the features from an existing image dataset

Transfer Learning process

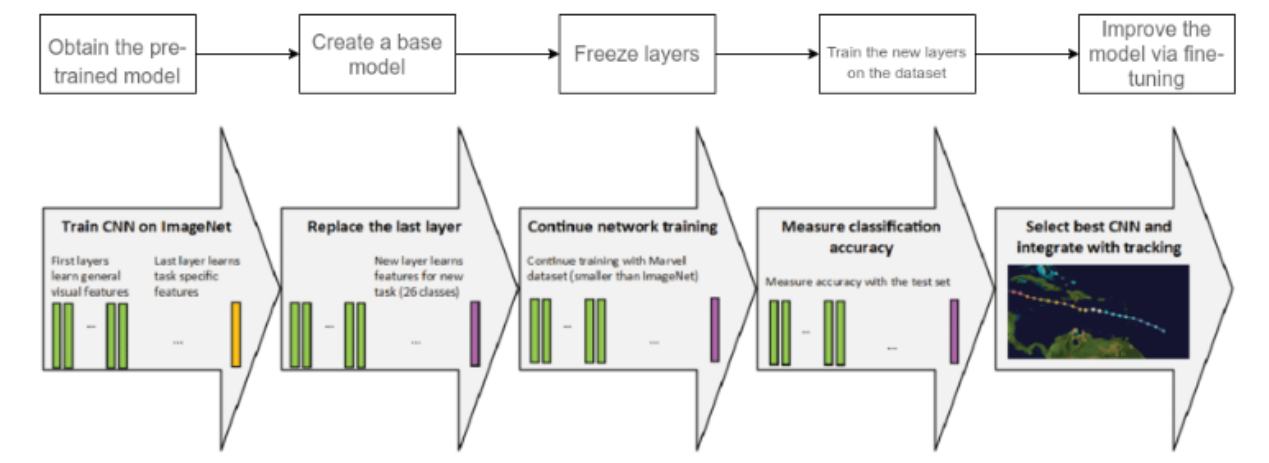


Figure 3.6 Transfer learning Steps (Leclerc, et al., 2018)

As shown in Figure 3.6, the Transfer Learning steps are:

1. Selecting a pre-trained model

There are several open-source models that have been trained on the ImageNet dataset and out of these, one that looks more suitable for the problem in question is selected (Krizhevsky, et al., 2012). The model choice depends on the image classes to be detected and whether these classes are part of the model's output layer. In this Thesis, Darknet-53 is a convolutional network and it is pre-trained on the ImageNet dataset. Darknet-53 is used as the foundation for object detection problems and YOLO workflows. This dataset can classify images into 1000 classes which makes it a very powerful tool. One of the classes is the traffic signal (but without the R, G, Y states) and Darknet-53 has been trained in this class.

2. Creating a base model

The architecture of Darknet-53 contains 53 convolutional layers with pre-trained weights. For creating the base model, the final output layer is removed and replaced by an output layer that is compatible with the problem in question. Figure 3.7 shows the Detection Flow Diagram.

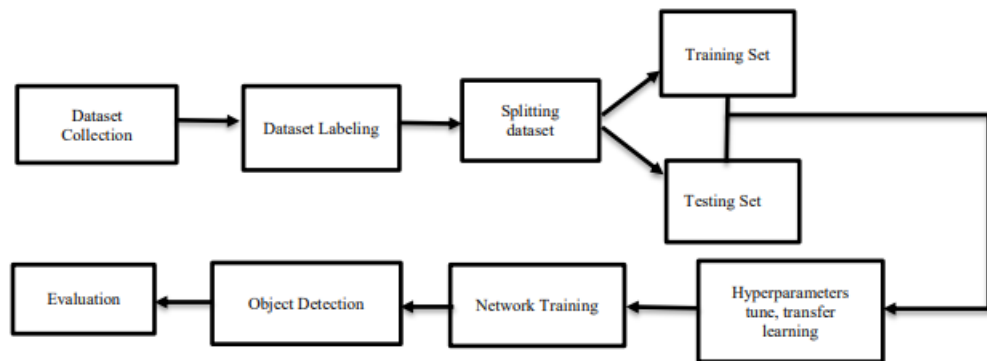


Figure 3.7 Detection Flow Diagram (Raza & Song, 2020)

Initially, each image of the new dataset is imported and passed through the already trained convolutional layers of Darknet-53 and its features extracted are stored. These features are inputs to the last trainable layer. The initial layers reflect general features, while the later trainable ones focus more on specific characteristics (see Fig. 3.7).

3. Locking layers so they don't change during training

This step is needed because the weights in these layers shouldn't be altered by training. The main idea is to keep the convolutional base in its original form

and then use its outputs to feed the classifier. The pre-trained model is used as a feature extraction mechanism that can be useful if either the computation power is low, the dataset is small or the pre-trained model solves a problem that is very similar to the problem in question.

4. Adding new “trainable” layers

This step adds new trainable layers that will turn extracted features into predictions for the new dataset.

5. Integrating with YOLOv3. Darknet53 is integrated with YoloV3. The objective of this integration is for YOLOv3 to localize the identified objects in the image (in this case the traffic signal) by placing the appropriate bounding boxes around these objects

This process is summarized in Fig. 3.8, where the integration of Darknet-53 with YOLOv3 is displayed (Benslimane, et al., 2019). The concept is the following: The layers of Darknet-53 are used with locked weights. The final (dense) layer of the network is modified to include in this case three classes (traffic light R, Y, G) instead of 1000. This part identifies the three states of the traffic light. Furthermore, YOLOv3 is integrated (at various layers) with Darknet-53 in order to locate the traffic lights in the photograph by fitting the appropriate bounding boxes.

It is noted that the integrated system of Figure 3.8 is adopted (downloaded), but its parameters are modified in order to fit the problem in question.

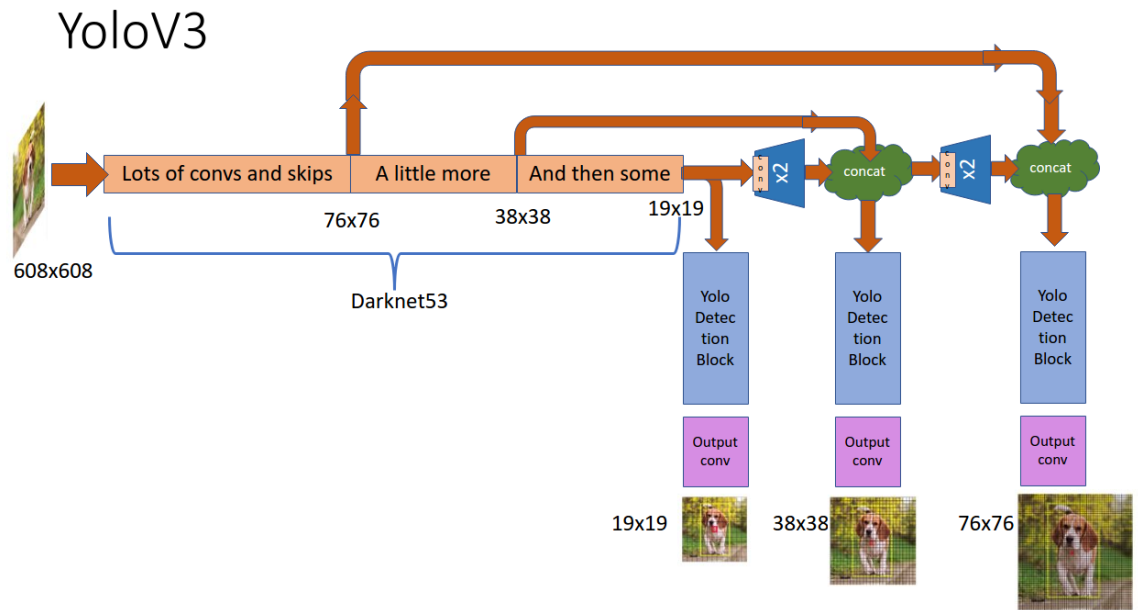


Figure 3.8 Adding of new “trainable” layers (Almog, 2020)

6. Training the trainable layers (last one of Darknet-53 and YOLOv3) using the new dataset

In this step, the additional layers of the model are trained.

The training process of the new model is analyzed in the Section 3.3.

3.3 Darknet-53+YOLOv3 model training

The training process of the model is described in the current Section. The training set up and the steps of the related algorithm are presented, and the model training results are analyzed. The final part of the Section includes the conclusions from the training exercise.

3.3.1 Setting up the model training environment

Section 3.2 described the transfer learning process and how the feature extraction mechanism of the pre-trained model can effectively help the new model produce better overall image classification results. Once all data of the dataset used for the training have been labelled, the actual training process of the trainable part of the model can begin. An adequate dataset size with correctly labelled objects plays an important role in the accuracy of the training process which -in the case of the work done as part of this Thesis- was based on the YOLOv3 Darknet model from AlekseyAB and was developed in the Google Collaboratory tool.

Google Colab is a product developed by Google Research. This online tool, which is accessible via a web-browser, helps data scientists and Artificial Intelligence researchers import existing software libraries and develop high level software code blocks which can then be shared online. Google Colab is especially well suited to machine learning, data science and analysis, and education. The environment allows anyone to write and execute python code with zero configuration through a web browser. Free access to computing resources including GPUs is also provided (Li, 2020).

Google Research also features Colab Pro, which provides faster Graphics Processing Units (GPUs), longer runtime limits and more memory allocation. Colab Pro also provides better connectivity options with the online server. The latter is useful in cases, in which large amounts of data are involved in the training process. Some of advantages and disadvantages between Colab and Colab Pro are shown in Table 3.2.

Note that the GPU is the core processing unit that implements the matrix multiplication operations involved in a Neural Network (NN). The higher the number of GPUs, the faster the NN data processing becomes and thus, state-of-the-art fast GPU processors make real-time CV algorithms possible (Abri, et al., 2020).

Table 3.1 Colab and Colab Pro tools (Buomsoo, 2020)

	Price	GPU	Runtime Limits
Colab	Free tool	K80: access a simple GPU card	A user can have up to 12 hours of run time
Colab Pro	Costs 10€/month as of June 2021	T4 & P100: access to high-end GPUs	A user can have up to 24 hours of runtime

For the purpose of the work done as part of this Thesis, the Google Colab Pro was chosen because of its higher processing power and memory limits. The software code taken from the GitHub software repository of AlexeyAB⁹ was used to train the NN model in the Colab Pro environment.

⁹ <https://github.com/AlexeyAB>

The algorithm comprises of the following steps as shown in Fig. 3.9:

Step 1: This first step activates the NVIDIA GPU controllers. CUDA is a parallel processing platform developed by NVidia that takes full advantage of the available GPU resources and is used in machine learning, gaming and deep learning applications. The CUDA platform is the Nvidia's language/API for programming the graphics card and is engineered to boost throughput in real-world applications (Rosebrock, 2020).

Step 2: This step accesses the custom dataset that includes the images to be used in the training and validation process. The dataset is stored online in the Google Drive service that is provided as part of the ¹⁰ (Google Workspace, 2012).

Step 3: This step utilizes the Darknet open-source neural network framework. Darknet is a network which can work together with the YOLO model as YOLO uses Darknet's pre-trained weights. For this process the network must first be compiled. Darknet requires that both the GPU and OpenCV¹¹ (see below) options be enabled. The CUDNN is a high level software library used by deep learning neural networks models and is built on the CUDA platform.

Step 4: This step creates a copy of the configuration file "yolov3.cfg" and names it "yolov3_training.cfg". The various configuration options that will be chosen for the custom model will be saved in the copy and not the original file. A backup of the original .cfg file is also kept for reference.

¹⁰ Google Drive is a cloud-based storage solution that allows you to save files online and access them anywhere from any smartphone, tablet, or computer.

¹¹ *"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code"* (OpenCV team , 2021).

Step 5: This step adds the necessary configuration options in the “yolov3_training.cfg” to define the number of classes of the objects to be classified. The parameters and their values are shown in Table 3.3 and are discussed below.

Table 3.2 Parameter modifications for training

Option name	Default value	Updated value	The relevant equation/comments
Batches (Batch size)	1	64	number of samples (e.g. images) which will be processed in one batch
Subdivision	1	16	Subdivision represents how the batch is again divided into blocks of images
Max batches	500200	6000	This equation ($Max_batches = 2000 \times n$) is used for 3 classes
Classes	80	3	3 classes are used for training (traffic light green, red and yellow)
Filters	255	24	The equation (Number of filters = $(n + 5) \times 3$) is used to calculate the filters used in the images
Learning rate	0.0001-0.1	0.0001-0.1	A learning rate parameter is a number between 0.0001 and 0.1 and controls how fast the values of weights change.
Steps	400000, 450000	4800, 5400	The learning rate is decreased after 4,800 and it is decreased much more after 5,400 iterations.
Scale			The scale parameter specifies how much the learning rate will decrease; thus, it multiplies the learning rate.

Batches

The batch size is changed from 1 (default) to 64 and refers to the number of photos that will be processed in one batch (the model loads 64 images that will be processed in each iteration).

Subdivision

Subdivision represents a further division of the batch into the blocks of images and it is changed from 1 (default) to 16. Starting with subdivisions = 1 there was an out of memory error. When we increased this parameter to 2, 4, 8, 16 etc. the training process started successfully. The GPU processes its images through batch and subdivision.

Max batches

The “Max Batches” configuration option defines the maximum number of batches, which training will run for. The number of “max batches” is reduced from 500,200 (default) to 6,000 because the classes are three instead of 80. The equation for “max batches” is the following:

$$Max_batches = 2000 \times n \quad (3.1)$$

where,

- n is the number of classes,
- 2000^{12} is a standard value from YOLOv3 (B.Sargunam & N.Kirthika, 2020)

Classes

There are three traffic lights classes (red, yellow and green). For this purpose, the 80 classes of the original file are changed to 3. Also, the number of filters is reduced from 255 (default value in the original file) to 24 that is:

$$\text{Number of filters} = (n + 5) \times 3 \quad (3.2)$$

where,

- n is the number of classes
- 3 represents the number of the bounding boxes used by YOLOv3
- 5 represents the 4 bounding box attributes plus one object confidence score.

Learning rate, steps and scale

The learning rate parameter is defined as a number between 0.0001 and 0.1 and controls how fast the values of weights change. At the beginning of the training

¹² <https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects>

process, the learning rate should be high. The learning rate decreases over time because as the network processes more data and converges towards the minimum of the loss function, the weights should change less aggressively. The step parameter is applied, which indicates that the learning rate will remain constant for many iterations and then will be decreased. This parameter must be 80% and 90% of the maximum batch value, which means that after $0.8 * \text{maximum batch iteration}$ (in our case $0.8 \times 6,000 = 4,800$), the learning rate will decrease and after a total of $0.9 * \text{maximum batch iterations}$ ($0.9 \times 6,000 = 5,400$), it will decrease further. The scale parameter specifies how much the learning rate will decrease.

Step 6: This step creates the `obj.name` and `.obj.data` files inside the `darknet/data/obj` directory. These files contain metadata information such as class names and number of classes required for the training process. The photos of the custom dataset are uploaded in two folders. The first folder is named “*train*” and contains the images for training while the second folder is named “*valid*” and contains the pictures for validation.

Step 7: The `yolov3_training.cfg` and the files which contain the class names of objects (`obj.names`) are copied to Google Drive.

Step 8: This step uploads the custom dataset in Google Drive in zip form and then “unzips” the photos file stored in Google Drive to the `darknet/data/obj` directory. There are two folders for unzipping. As mentioned above, the “*valid*” folder contains the photos for the validation process, the “*train*” folder contains the photos used in the training process.

Step 9: This step creates two files in `.txt` form, named `train` and `valid`, respectively. The folders contain the location with the last part containing the names of all images (e.g. `/content/gdrive/MyDrive/yolov3/darknet/data/obj/train/out00000.png`). The images will be fetched from the location specified in this file during training.

Step 10: In this step the pre-trained weights of Darknet-53 are downloaded for the convolutional layers. In Section 3.2, Transfer Learning is discussed as well as how Darknet-53 interacts with YOLOv3.

Step 11: In this step, the pre-trained weights are loaded into the YOLOv3 model and the training process begins. The model takes about 8-10 hours to train for 3 classes. The time required for the training of a custom model depends on the dataset size and the number of classes. We used one GPU resource (Nvidia Tesla P100-PCIE-16GB), from Google Colab Pro with a speed of 32 GB/sec. In case the training process stops (due to network or power failure or non-availability of GPU resource allocation), it can start again and continue from the last saved weights.

The steps and the high-level software code are presented in Fig. 3.9 below:

Step 1: Check if NVIDIA GPU is enabled

```
!nvidia-smi
```

Step 2: Mount your Google Drive on Google Colab.

```
from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/
!ls /content/gdrive/MyDrive/yolov3
```

Step 3: Configure and compile Darknet.

Configure

```
%cd /content/gdrive/MyDrive/yolov3/darknet
#!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
```

Compile

```
!make
```

Step 4: Make a copy of yolov3.cfg

```
!cp cfg/yolov3.cfg cfg/yolov3_training.cfg
```

Step 5: Change lines in yolov3.cfg file

```
!sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
!sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 6000/' cfg/yolov3_training.cfg
!sed -i '610 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=24@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=24@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=24@' cfg/yolov3_training.cfg
!sed -i '22 s@steps=400000,450000@steps=4800,5400@' cfg/yolov3_training.cfg
```

Step 6: Create .names and .data files.

```
!echo -e 'traffic light green\ntraffic light red\ntraffic light yellow' > data/obj.names
!echo -e 'classes= 3\ntrain                               = data/train.txt\nvalid
= data/test.txt\nnames = data/obj.names\nbackup = /content/gdrive/MyDrive/yolov3' > d
ata/obj.data
```

Step 7: Save yolov3_training.cfg and obj.names files in Google Drive.

```
!cp cfg/yolov3_training.cfg /content/gdrive/MyDrive/yolov3/yolov3_testing.cfg
!cp data/obj.names /content/gdrive/MyDrive/yolov3/classes.txt
```

Step 8: Unzip the images dataset.

```
!mkdir data/obj
!unzip /content/gdrive/MyDrive/yolov3/train.zip -d data/obj/train

!mkdir data/obj/test
!unzip /content/gdrive/MyDrive/yolov3/valid.zip -d data/obj/valid
```

Step 9: Create train.txt file.

```
import glob
images_list = glob.glob("/content/gdrive/MyDrive/yolov3/darknet/data/obj/train/*.jpg")
with open("data/train.txt", "w") as f:
    f.write("\n".join(images_list))
```

```
images_list = glob.glob("/content/gdrive/MyDrive/yolov3/darknet/data/obj/valid/*.jpg")  
with open("data/test.txt", "w") as f:  
    f.write("\n".join(images_list))
```

Step 10: Download pre-trained weights for the convolutional layers file.

```
!wget https://pjreddie.com/media/files/darknet53.conv.74
```

Step 11: Start training.

```
!./darknet detector train data/obj.data cfg/yolov3_training.cfg darknet53.conv.74 -  
dont_show -map | tee output.log
```

Figure 3.9 Code for developing and training the traffic signal detection model

3.3.2 Basic of model training aspects

There are two important metrics used in training and validation. The training loss is used to measure the error between predicted and true values related to the bounding box and ground truth bounding box respectively. In addition, the training loss is used to assess the training process. The mean average precision is an accuracy metric that shows how accurate the model is. It is particularly useful in validation.

In the graphs used to present the results of the experiments below two lines are shown. The blue line represents the training loss (related to the training dataset) and the red line represents mAP which is related to the validation dataset.

3.3.3 Training Loss

The loss function in YOLOv3 consists of three parts:

1. Localization loss (error between the predicted bounding box and ground truth bounding box)
2. Confidence loss
3. Classification loss

These three parts are related to the following errors.

That is (Wu & Xu, 2020).

$$Loss = Error_{coord} + Error_{iou} + Error_{cls} \quad (3.3)$$

where,

1. $Error_{coord}$: refers to the coordinate prediction error (localization loss).
2. $Error_{iou}$: refers to an Intersection Over Union(IoU) error (confidence loss)
3. $Error_{cls}$: refers to the classification error or loss

Localization Loss (coordinate prediction error)

Localization loss assesses the errors between bounding box center coordinates and ground truth box center coordinates.

$$Error_{coord} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\ + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \quad (3.4)$$

where,

- λ_{coord} : is a weight parameter
- S^2 : refers to the number of the grids ($S \times S$). For example, in 13×13 grid the S can take values from $i = 0, 1, 2, \dots, 12$.
- 1_{ij}^{obj} : is 1 if an object has been detected in the j th bounding box in cell i , otherwise it is 0. This parameter refers to whether there is an object that falls in the j th bounding box of the i th grid cell
- $(\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i)$ refer to the predicted bounding box parameters (center coordinates, width and height)
- (x_i, y_i, w_i, h_i) refer to the center coordinates, width and height of the ground truth box
- $\sum_{j=0}^B$: this sum is calculated for each anchor box (5 in total), where $B=5-1=4$ (because the index starts from 0)
- B : refers to the number of bounding boxes per grid cell.

As one can see in the Equation, the square root of the bounding box width and height is used in the calculation of the localization loss. This means that small prediction deviations from the actual box matter less in large boxes than in small boxes.

Confidence Loss (IoU error)

The error associated with the IoU score is calculated by:

$$\begin{aligned} Error_{iou} = & \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} (c_i - \hat{c}_i)^2 \\ & + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{noobj} (c_i - \hat{c}_i)^2 \end{aligned} \quad (3.5)$$

where,

- c_i : represents the confidence score. This is a metric that indicates the probability that the bounding box predicted by the model actually contains the object
- \hat{c}_i : represents the intersection over union of the predicted bounding box with the ground truth bounding box.
- 1_{ij}^{obj} : is 1 when an object is detected in the j th bounding box of cell i , otherwise it is 0.
- 1_{ij}^{noobj} When there is an object present in the cell the value is 0. When there is no object in the cell the value is 1.
- λ_{noobj} : equals to 5

Classification Loss

$$Error_{cls} = \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \quad (3.6)$$

where,

- 1_i^{obj} : takes the value of 1 if an object is present in cell i , otherwise it is 0
- $\hat{p}_i(c)$: represents the probability of the object detected in cell i to actually belong to class c
- S^2 refers to the number of the grids ($S \times S$)

- $p_i(c)$: should be 1 if the object in cell i belongs to class c and 0 otherwise.

The total Loss value of the NN is the sum of classification loss, localization loss and confidence loss (see eq.3.3) (Ahmad, et al., 2020) (Gai, et al., 2021).

$$\begin{aligned}
 Loss &= Error_{coord} + Error_{iou} + Error_{cls} \\
 &= \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
 &\quad + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
 &\quad + \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{obj} (c_i - \hat{c}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B 1_{ij}^{noobj} (c_i - \hat{c}_i)^2 \\
 &\quad + \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
 \end{aligned} \tag{3.7}$$

3.3.4 Mean Average Precision (mAP)

As mentioned above, this metric is used to evaluate the performance of object detection models such as YoloV3, R-CNN and SSD Average Precision (AP).

Two equations are used in computing AP. One is the precision equation (3.8) and the other is the Recall equation (3.9). Precision is a percentual metric to assess how accurate the predictions are. Recall is a percentual metric of the number of positive predictions made (Gai, et al., 2021).

$$Precision = \frac{TP}{TP + FP} \tag{3.8}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.9}$$

where,

- TP represents the true positive (Predicted as positive and was correct)
- FP represents the false positive (Predicted as positive but was incorrect)
- FN Represents the false negative (Failed to predict an object that was there)

3.3.5 Results of training

The model described in Section 2.3 was trained using the data described in Section 3.1. Several training runs were performed using combinations of the datasets of Section 3.1. The objective is to optimize training by selecting those parameters that result to the best possible validation.

Thus, image datasets were separated into two subsets: The first contained the images used for training and the second the images used for validation. The training subset is used to build the model. The validation data are used to evaluate the model using the accuracy metric of mean average precision. To assist in distinguishing the datasets easily we renamed them as in the following Table.

Table 3.3 Upgrade names of datasets

Original names	Upgrade names
Large Dataset	CARLA Dataset
SJTU Small Traffic Light Dataset	Clear Dataset
Combination of Bosch Small Traffic Lights Dataset and Berkley DeepDrive Dataset	Blurred Dataset

1st Training exercise

This exercise uses images from Carla’s simulator (as already discussed in Section 3.1). From 1,000 total photos 700 were used for training and 300 for validation.

Figure 3.10 shows the results of the training process of the model. The training loss refers to the training process that uses the training dataset and mAP to the validation process that uses the validation dataset.

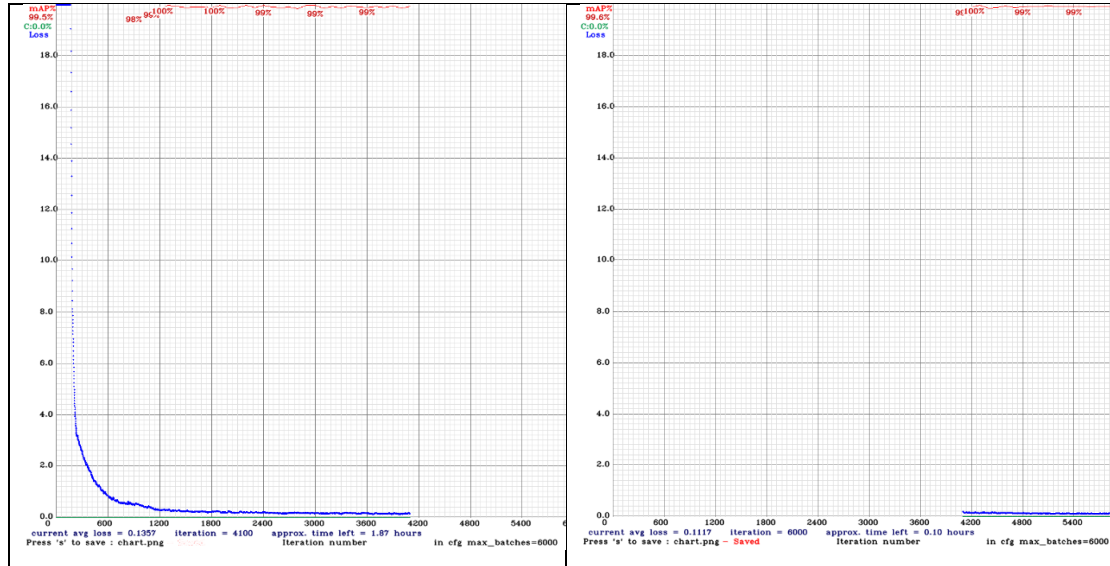


Figure 3.10 Results of Carla dataset (training loss and mAP)

In the initial iterations, both the training loss (blue line in Fig. 3.10) and the learning rate (slope of blue line) are high as expected, since the “trainable” layers of the model contain just the initialization weights. Subsequently, the training loss falls as iterations increase. Between 1800 and 6000 iterations, the training loss falls close to zero and the mAP increases to 99.5%. This is expected because the photos used in the training process contained clear images of traffic lights taken from a close range. The high mAP is attributed to the clarity of the images used in the custom dataset. The right image follows the left one and shows the training process from 4,000 to 6,000 iterations.

2nd Training exercise

The second dataset contained 1,217 clear traffic light images within a generic environment that contains also other objects (road elements, signs, vehicles, etc.) from the “SJTU Small Traffic Light” dataset. Out of the 1,217 images, 852 images were used for training and 365 images were used for validation.

The mean average precision for this dataset is shown in Fig. 3.11 and came out to be 38.5%. The mAP value is very low because the number of photos (dataset) is limited.

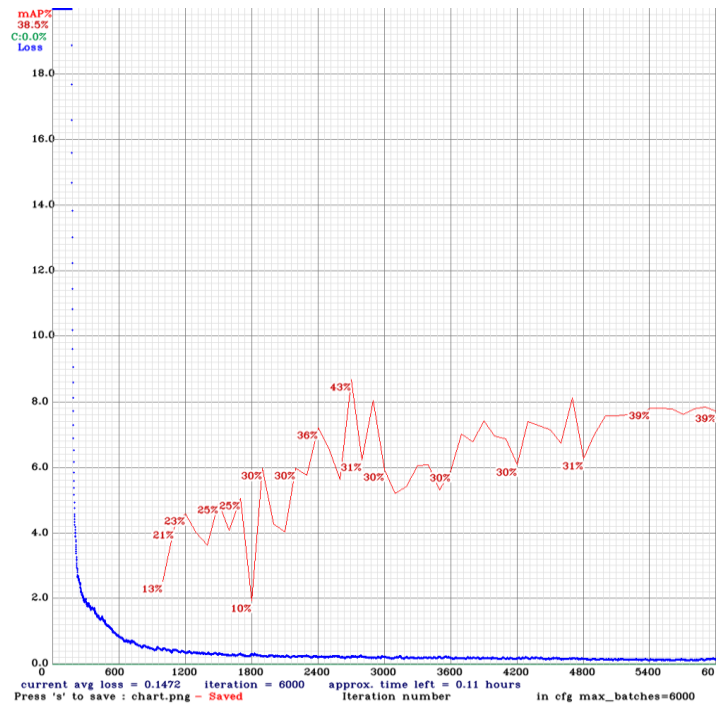


Figure 3.11 Results of training loss and mAP for Clear dataset

3rd Training exercise

The third experiment contains 3,455 images out of which 2,419 images are used for training and 1,036 images are used for validation. As already discussed in Section 3.1, this dataset contains images under various conditions such as day or night etc. and many blurred ones.

The results of this dataset are shown in Figure 3.12. The mAP is 41.7%. The fluctuation in mAP may be explained by the fact that some batches contain more blurred pictures than the others.

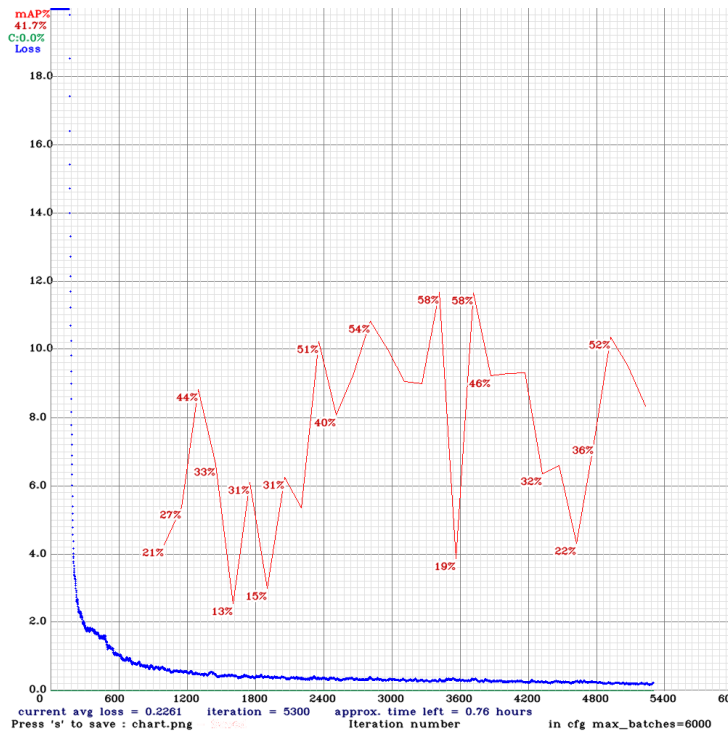


Figure 3.12 The training loss and mAP of Blurred dataset

4th Training exercise

The fourth experiment contains all the three datasets. In this experiment, the total number of images is 5,672 as presented in Table 3.5 :

Table 3.4 The combination of three datasets

Datasets	All (First+Second+Third)	CARLA	Clear	Blurred
Training set (70%)	3971	700	852	2419
Validation set (30%)	1701	300	365	1036
Total	5672	1000	1217	3455

The results are shown in Fig. 3.13. The mAP is 70,7%. The drop near the 4000th iteration possibly occurs when the code reads batches that contain the blurred photographs. This has a negative effect on the final prediction. In this case as well mAP fluctuates depending on the images contained in each batch.

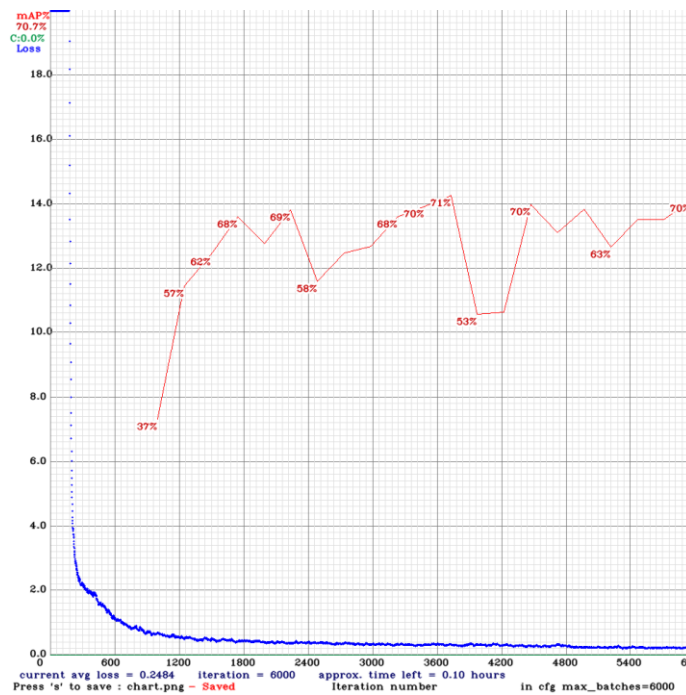


Figure 3.13 Results of the combination of three datasets

5th Training exercise

In this case, the Carla and Clear datasets are used, that is in total, 2,217 images of which 1,552 are used for training (700 from the CARLA dataset and 852 the Clear dataset). 665 images are used for the validation process (300 from CARLA and 365 from Clear).

Table 3.5 Combination of CARLA and Clear datasets

Datasets	First and second	CARLA	Clear
Training set (70%)	1552	700	852
Validation set (30%)	665	300	365
Sum	2217	1000	1217

The results are shown in Fig. 3.14. The mAP is 76.8%. The dataset includes a large number of photos and many clear ones.

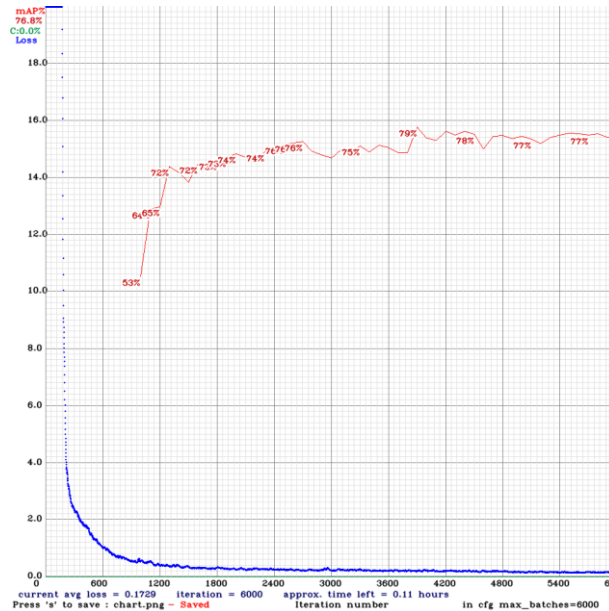


Figure 3.14 Results of combination of *CARLA* and *Clear* datasets

6th Training exercise

The sixth experiment contains two datasets: CARLA + Blurred. In this experiment, the total number of images is 4,455 as presented in Table 3.7 :

Table 3.6 Combination of CARLA and Blurred datasets

Datasets	First and third	CARLA	Blurred
Training set (70%)	3119	700	2419
Validation set (30%)	1336	300	1036
Sum	4455	100	3455

The results of this dataset are shown in Figure 3.15. The mAP is 79,1%. The mAP fluctuates, possibly due to the blurred images.

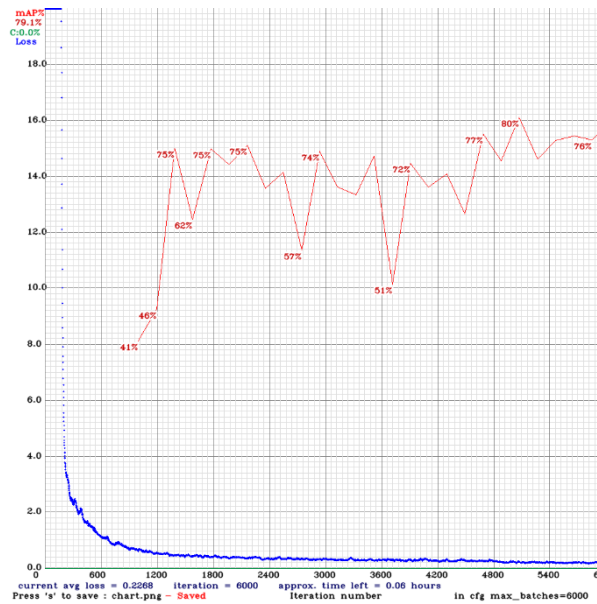


Figure 3.15 Results of combination of *CARLA* and *Blurred* datasets

7th Training exercise

The final experiment contains two datasets: Clear and Blurred. In this experiment, the total number of images is 4,672 as presented in Table 3.8:

Table 3.7 Combination of Clear and Blurred Datasets

Datasets	Second and third	Clear	Blurred
Training set (70%)	3271	852	2419
Validation set (30%)	1401	365	1036
Sum	4672	1217	3455

The mean average precision for this dataset is shown in Fig. 3.16 and came out to be 48,7%. The mAP is very low.

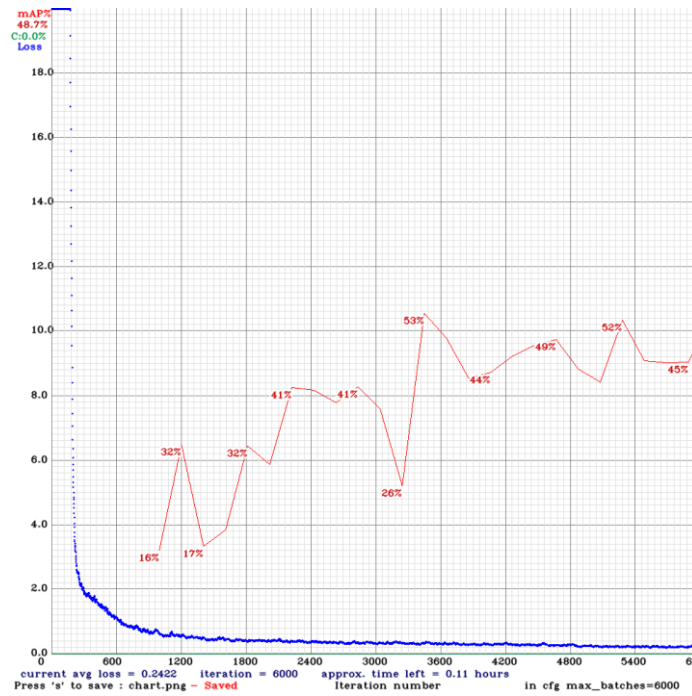


Figure 3.16 Results of combination of Clear and Blurred datasets

Concluding Remarks

Table 3.9 presents a synopsis of the above experiments.

Table 3.8 Synopsis of the experiments

Experiments	Datasets	Mean Average Precision (mAP)	Average loss
1	CARLA	99.5%	0.13
2	Clear	38.5%	0.15
3	Blurred	41.7%	0.23
4	CARLA + Clear+ Blurred	70.7%	0.25
5	CARLA + Clear	76.8%	0.17
6	CARLA + Blurred	79,1%	0.23
7	Clear and Blurred	48,7%	0.24

The results in Table 3.9 show that the highest mAP results always include the CARLA dataset and thus, the CARLA dataset helps the training process significantly. This is because the photographs of the CARLA dataset display traffic lights very clearly. The Blurred dataset also helps the training process due to the increased number of images

in the dataset. The combination of CARLA and Blurred datasets gives a strong result in the training process.

Chapter 4 Traffic light state detection: A case study

In the previous Chapter we described the NN model and how it has been trained. The current Chapter deals with the performance of the trained model and its refinement on real-life traffic light photos taken in a general urban environment under various conditions. These photos were frames extracted from videos and contain multiple background objects in addition to the traffic lights. The latter assumed all three states (red, green, yellow).

4.1 Experimental set up

The images that were used for testing were created from a compilation of small videos with a total duration of 77 minutes. These videos were taken by an iPhone camera from inside a moving car while driving along a route that contained a high number of traffic lights. The images contain scenes with traffic lights, roads and cars and were taken in different times of the day. A total of 165 small duration videos were taken as part of the experiment and contain the identity and the state of each traffic light. The videos were then converted to 2,817 still images which were used as input to the model. The total number of traffic lights contained in all videos were 182 (a video may contain more than one traffic light in some cases). Of these, 97 videos contain all three states of the traffic light (green, red, yellow), whereas 85 videos contain only one state, green. This is because it was practically difficult to wait for the traffic light to change state due to traffic.



Figure 4.1 State of 1st and 2nd from 180 traffic lights

In Fig. 4.1 the first 3 images show the first traffic light (out of the 180 contained in the dataset) in all 3 states and the bottom image shows the second traffic light only in its red state, the only one existing in the dataset. This Figure illustrates that some traffic lights have been photographed in only one state and some have been photographed in all 3. In total, 305 traffic light and traffic light state combinations were created.

4.2 Data processing

The network architecture of the model used in processing the above photographs is the one described in Section 3.3. Concerning the weights, we Table 4.1 shows the training sessions used to obtain the alternative weights values used in the case study.

Table 4.1 Datasets used to obtain alternative weights

Training session	Dataset used
1	Carla and clear
2	Carla and blurred
3	Carla
4	Clear and blurred

For tuning the model parameters and processing the test images / photos, we used the four steps described below and presented in the code of Fig. 4.2:

Step 1: The images of the custom dataset are uploaded to a cloud storage space (Google Drive) in compressed (.zip) form. This step also uncompresses the images from the cloud storage and places them to the darknet/data/obj folder directory.

Step 2: The physical folder and file location of the images is saved and this step creates one text file (txt form), named “test”, which contains the location of all images.

Step 3: Since it was necessary to adapt the code to our own set of data, some changes were made to the .cfg file. As mentioned in Section 3.3, the number of subdivisions and the number of batches affect the outcome. In this case the number of batches was changed from 64 to 1 and the number of subdivisions from 16 to 1. The reason for changing these parameters is to test 2,817 images one by one. The output of the model contains the results of the test process (i.e. the recognition of the traffic lights) for each image separately.

Step 4: This step tests the weights from the initial training datasets on the custom data and the results are checked for accuracy. The model has been set up for testing and the testing process takes around 6-7 minutes for each weight case in each batch. The time required for testing the custom model depends on the dataset size and the

number of classes (3 in this case). This last step outputs the photos used in the test process together with the traffic light recognition results.

Step 1:

```
!mkdir data/obj/test
!unzip /content/gdrive/MyDrive/yolov3/test.zip -d data/obj/test
```

Step 2:

```
import glob
images_list = glob.glob("/content/gdrive/MyDrive/yolov3/darknet/
data/obj/test/*.jpg")
with open("data/test.txt", "w") as f:
    f.write("\n".join(images_list))
```

Step 3:

```
%cd cfg
!sed -i 's/batch=64/batch=1/' yolov3_training.cfg
!sed -i 's/subdivisions=16/subdivisions=1/' yolov3_training.cfg
%cd ..
```

Step 4:

```
!./darknet detector test data/obj.data cfg/yolov3_training.cfg /
content/gdrive/MyDrive/yolov3/weights/Carla_and_blurred_images/y
olov3_training_final.weights -
dont_show < /content/gdrive/MyDrive/yolov3/darknet/data/valid.tx
t > result.txt
```

Figure 4.2 Test data processing

4.3 Results of the Neural Network Model

The results of processing the photographs of the Thessaloniki dataset by the model were classified into three categories (True predictions, False predictions, No predictions).

Table 4.2 shows the predictions made by each of the four models.

Table 4.2 Results of predictions

weights	True Prediction	False Prediction	No prediction
Carla and clear (pictures)	1,261	203	1,353
percentage	45%	7%	48%
Carla and blurred (pictures)	1,706	135	976
percentage	61%	5%	34%
Carla (pictures)	952	84	1,781

percentage	34%	3%	63%
Clear and blurred (pictures)	1,712	78	1,027
percentage	61%	3%	36%
max	61%	7%	64%

As shown in the Table above, not all weights performed well. Many of the False or No predictions were encountered in the sets containing the photographs taken in low light conditions (evening/night). In these cases, the model confuses the traffic lights with the stop lights of the vehicles in the photo. An example of this is shown in Figure 4.3. In the photo on the left there are traffic lights which were not identified and in the photo on the right, no traffic lights were identified. In both cases the model is confused by the car rear lights.

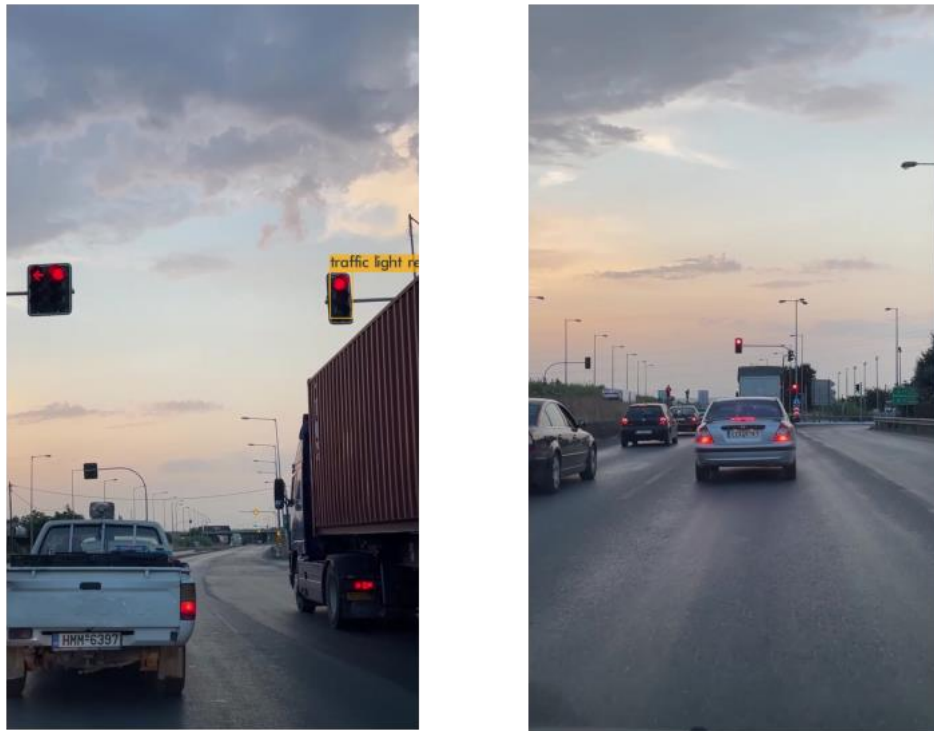


Figure 4.3 False prediction results

The highest percentage (61%) in True Predictions correspond to the weights of “Carla and Blurred” and “Clear and Blurred” datasets. This is because the weights were determined using a large number of photographs. In the case of “Carla and Blurred” the Carla images were from simulated traffic lights. This may be the reason why the

percentage of False Predictions (5%) is higher. On the other hand, the “Clear and Blurred” dataset contains multiple real-life photographs in various conditions taken from various distances and the False Predictions are lower. Furthermore, the photos also contained other objects such as cars, road, etc. In this respect, the “Clear and Blurred” dataset fits well to the Thessaloniki dataset and that's why its weights are able to make the best predictions and object detection in the test photographs.

In general, however, the results have a low percentage of True Predictions. This is attributed mostly to the low number of images in the datasets used to train the model.

4.4 Improving the Results of the Neural Network Model

We used a simple idea in order to improve the results of the model:

- Consider a combination of a traffic light and its state for which we had multiple photos. For each photo the model recognized one of the three possible states, or did not recognize any state
- The state attributed to this combination of traffic light - state results from the majority between the above recognitions, ignoring the No Predictions.

That is, if we had 6 photos for such a combination, in 4 of which a red signal was recognized, in 1 photo a green signal was recognized, and in 1 no signal/state was recognized, then the state attributed to the combination was red. Consider another example with 5 photos: 2 green, 1, yellow, 3 No recognition. The state attributed was green. In the event of a tie, the state was labelled as inconclusive.



Figure 4.4 States of 10th Traffic light

Consider the traffic light of Figure 4.4, which is contained in multiple video frames (33 video frames). Out of all these video frames 4 frames were taken in the green state, 28 were taken in the red state and was taken in the yellow state.

- Out of the 4 photos of the green state, the 2 photos were indeed detected by the model as green (True Prediction) but the other 2 were detected as red (False Prediction). So, this traffic light-state prediction was classified as **Inconclusive**.
- Out of the 28 red state photos, 7 photos were not detected by the model (No Prediction) and these were excluded from the majority calculation. Consequently, the traffic light was categorized based on the majority of the remaining photos. 21 the remaining photos with recognized as red. Thus, the prediction for this traffic light was classified under the **True Predictions** category.

Table 4.3 shows the percentage of predictions of traffic light-state combinations of the two best models.

Table 4.3 Improvement method results

trained weights	True Prediction	False Prediction	No prediction	Inconclusive
Carla and blurred	239	13	52	1
(%)	78%	4%	17%	0%
Clear and blurred	237	13	50	5
(%)	78%	4%	16%	2%

As shown in Table 4.3, the models with weights from the Carla and blurred dataset predict the correct outcome with 78% accuracy (True prediction), predict an incorrect state in 4% of the cases and cannot predict a state in 17% of the cases. On the other hand, the weights from the “Clear and Blurred” dataset predict the correct outcome with 78% accuracy (True prediction), predict incorrectly a state in 4% of the cases and cannot predict a state in 16% of the cases. These results demonstrate that

- The refinement method improves the True Prediction percentages significantly from 61% to 78%. This is due to the fact that the majority rule reduces significantly the No Prediction percentage
- Both models perform equally well in the case of traffic light-state predictions.

4.5 Conclusion

A high-accuracy and real-time object detection algorithm is part of the safety and real-time control systems of autonomous vehicles. Various studies focus on safety of autonomous driving and describe models which provide satisfactory -but not perfect- predictions. This is due to the trade-off between accuracy and the model’s operational speed. For this reason, this study proposes an object detection algorithm (Darknet-53+YOLOv3) that achieves a reasonable trade-off between accuracy and operational speed when trained in specific datasets. A high rate (78%) of traffic light states were correctly predicted and only 5% of the states were predicted incorrectly. To this effect, the Darknet-53+YOLOv3 algorithm combined with the majority rule may significantly improve detection and accuracy.

Chapter 5 Conclusions

We propose an accurate and effective model that can detect traffic lights and their states in real-time. Such a model can be practically used in systems that control autonomous vehicles. The implementation is based on the Darknet-53+YOLOv3 model. Two main processes are analyzed: the training process which “trains” the model to detect the traffic lights using existing image datasets, and the testing process which tests the effectiveness of the model in detecting traffic light states in images collected from numerous traffic lights of Thessaloniki.

For the training process we used three image datasets and combinations of them. During this process, we conducted experiments for each dataset and each combination and compared the detection accuracy of the model and its resulting weights. The image dataset which proved to be the best for training the model was a combination that contains a) clear photographs of traffic lights taken from a close distance, b) clear photographs of traffic lights within a general environment, and c) blurred photographs of traffic lights taken from a distance under various light conditions, in the presence of objects that resemble (but are not) traffic lights. In addition, as expected, the volume of the images in the dataset and the light condition in which the photographs were captured, influence the results of the process. The results of the training process show that the highest mAP results always include the CARLA dataset and thus, the CARLA dataset helps the training process significantly.

For the test process we used a custom image dataset that was created by the author and contains photographs of multiple traffic lights (under various states) taken from Thessaloniki streets. The test was performed using the most appropriate Neural Network weights that were obtained during the training process of the model.

The testing results indicated a 61% percentage of true predictions, with 3% false predictions and 36% no predictions. Using a simple post-processing step that is based on the majority of predictions among multiple photographs of a certain traffic light

and state, the no prediction percentage fell significantly to 16%, while the true prediction accuracy was improved to 78%.

As a result, the proposed approach can significantly improve the camera-based object detection system in autonomous vehicles. This technique of traffic lights detection can serve as a prototype for future development.

Recommendations for future work include the following:

- Utilize different networks (except Darknet-53) to be used in conjunction with YOLOv3 for object detection (for example, Darknet19, Resnet 101, etc.)
- Use different models (except YOLOv3) for object localization (e.g. R-CNN, Faster R-CNN, SSD, etc.)
- Increase the number of images in the training process
- Experiment with different parameters of the model (through the configuration file)
- Apply other pre-trained weights from other datasets include different pictures

References

- Abri, S., Çetin, S., Yarıcı, A. & Abri, R., 2020. Multi-Thread Frame Tiling Model in Concurrent Real-Time Object Detection for Resources Optimization in YOLOv3. *ICCTA '20: Proceedings of the 2020 6th International Conference on Computer and Technology Applications*, 14 April, pp. 69-71.
- B.Sargunam & N.Kirthika, 2020. Detecting Multi-Class Artifacts in Endoscopic Images using YOLOv3. *International Journal of Recent Technology and Engineering (IJRTE)*, 26 May, 9(1).
- Benslimane, S., Tamayo, S. & de La Fortelle, A., 2019. Classifying logistic vehicles in cities using Deep learning. *WORLD CONFERENCE ON TRANSPORT RESEARCH SOCIETY*, 4 June, pp. 6-7.
- Choi, J., Chun, D., Kim, H. & Lee, H.-J., 2019. Gaussian YOLOv3: An Accurate and Fast Object Detector Using Localization Uncertainty for Autonomous Driving. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 9 April, pp. 1-10.
- Kozel, R. & Robert, N., 2020. Real-Time Traffic Light Identification using YOLOv3 Algorithm For Autonomous Vehicles. *CAT Vehicle 2020 Technical Reports*, 12 April, pp. 0-10.
- Raza, K. & Song, H., 2020. Fast and Accurate Fish Detection Design with Improved YOLO-v3 Model and Transfer Learning. *International Journal of Advanced Computer Science and Applications*, January, 11(2), pp. 7-11.
- Wang, S., 2021. Yolo-Series Algorithms: Comparison and Analysis of Object Detection Models for Real-Time UAV Applications. *IOP Publishing Ltd*, 14-16 May, 1948(1).
- Wu , Y. & Xu, D., 2020. Improved YOLO-V3 with DenseNet for Multi-Scale Remote Sensing Target Detection. *Computer Science, Medicine*, 31 Jule, pp. 10-13.
- Aakash, N., Sayak, P. & Maynard-Reid, M., 2021. *Keras: Dataset preprocessing*. [Online]
Available at: <https://keras.io/api/preprocessing/>
[Accessed 2021].
- Ahmad, T. et al., 2020. Object Detection through Modified YOLO Neural Network. *Scientific Programming*, 6 June, pp. 2-4.
- Almog, U., 2020. *YOLO V3 Explained*. [Online]
Available at: <https://towardsdatascience.com/yolo-v3-explained-ff5b850390f>

Balla, N., 2020. *FIVE POPULAR DATA AUGMENTATION TECHNIQUES IN DEEP LEARNING*. [Online]

Available at: <https://dataaspirant.com/data-augmentation-techniques-deep-learning/>

Boesch , G., 2021. *Object Detection in 2021: The Definitive Guide*. [Online]

Available at: <https://viso.ai/deep-learning/object-detection/>

Boesch, G., 2021. *What is Image Annotation? (Easy-to-understand Guide)*. [Online]

Available at: <https://viso.ai/computer-vision/image-annotation/>

Brain, G., 2021. *tf.data: Build TensorFlow input pipelines*. [Online].

Buomsoo, K., 2020. *Google newly launches Colab Pro! - comparison of Colab and Colab pro*. [Online]

Available at: <https://buomsoo-kim.github.io/colab/2020/03/15/Google-newly-launches-colab-pro.md/>

[Accessed 2021].

Deng, J. et al., 2009. ImageNet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, 20-25 June, pp. 1-2.

Duerig, T. & Krasin , I., 2016. *Introducing the Open Images Dataset*. [Online]

Available at: <https://ai.googleblog.com/2016/09/introducing-open-images-dataset.html>

Elisha, O., 2020. *Overcoming overfitting in image classification using data augmentation*. [Online]

Available at: <https://heartbeat.fritz.ai/overcoming-overfitting-in-image-classification-using-data-augmentation-9858c5cee986>

Gai, W., Liu, Y., Zhang, J. & Jing, G., 2021. An improved Tiny YOLOv3 for real-time object detection. *Systems Science & Control Engineering*, 23 March, 9(1), pp. 314-321.

Gandhi, A., 2021. *Data Augmentation / How to use Deep Learning when you have Limited Data*. [Online]

Available at: <https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/>

Ganesh, P., 2019. *Towards data science*. [Online]

Available at: <https://medium.com/@prakhargannu>
[Accessed 10 07 2021].

Girshick, R. D. J. D. T. a. M. J., 2014. *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation..* Columbus, Ohio., 2014 IEEE Conference on Computer Vision and Pattern Recognition.

Goodfellow, I., Bengio, Y. & Courville, A., 2016. *Deep Learning*. s.l.:The MIT Press.

- Malevé, N., 2019. *An Introduction to Image Datasets*. [Online]
Available at: <https://unthinking.photography/articles/an-introduction-to-image-datasets>
- Mantripragada, M., 2020. *Digging deep into YOLO V3 - A hands-on guide Part 1*. [Online]
Available at: <https://towardsdatascience.com/digging-deep-into-yolo-v3-a-hands-on-guide-part-1-78681f2c7e29>
- Mohana, H. V. R. A., 2019. Object Detection and Classification Algorithms. 8 June.
- Morgunov, A., 2021. *Deep Learning Guide: Choosing Your Data Annotation Tool*. [Online]
Available at: <https://neptune.ai/blog/annotation-tool-comparison-deep-learning-data-annotation>
- Mwiti, D., 2021. *Transfer Learning Guide: A Practical Tutorial With Examples for Images and Text in Keras*, s.l.: s.n.
- Novak, B., Ilic, V. & Pavkovic, B., 2020. YOLOv3 Algorithm with additional convolutional neural network trained for traffic sign recognition. *2020 Zooming Innovation in Consumer Technologies Conference (ZINC)*, 24 June.pp. 1-5.
- OpenCV team , 2021. *opencv*. [Online]
Available at: <https://opencv.org/about/>
[Accessed August 2021].
- Redmon, J. D. S. G. R. a. F. A., 2016. *You Only Look Once: Unified, Real-Time Object Detection*. Las Vegas, Nevada, United States., s.n.
- Redmon, J. & Farhadi, A., 2018. YOLOv3: An Incremental Improvement. *Computer Science*, 8 April, pp. 1-6.
- Ren, S. H. K. G. R. a. S. J., 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- Rosebrock, A., 2019. *Keras ImageDataGenerator and Data Augmentation*. [Online]
Available at: <https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/>
- Rosebrock, A., 2020. *How to use OpenCV's "dnn" module with NVIDIA GPUs, CUDA, and cuDNN*. [Online]
Available at: <https://www.pyimagesearch.com/2020/02/03/how-to-use-opencvs-dnn-module-with-nvidia-gpus-cuda-and-cudnn/>
[Accessed August 2021].

Sarin, S., 2019. *Exploring Data Augmentation with Keras and TensorFlow*. [Online]
Available at: <https://towardsdatascience.com/exploring-image-data-augmentation-with-keras-and-tensorflow-a8162d89b844>

Saxena, P., 2021. YOLO : You Only Look Once – Real Time Object Detection. 29 January.

Sharma, P., 2018. *A Practical Guide to Object Detection using the Popular YOLO Framework – Part III (with Python codes)*. [Online]
Available at: <https://www.analyticsvidhya.com/blog/2018/12/practical-guide-object-detection-yolo-framework-python/>

Srivastana, V., 2017. *Self Driving Vehicles: Traffic Light Detection and Classification with TensorFlow Object Detection API*. [Online]
Available at: <https://becominghuman.ai/traffic-light-detection-tensorflow-api-c75fdbadac62>

Tepteris , G. V., 2020. *Autonomous Vehicles: Basic Concepts in Motion Control and Visual Perception*, s.l.: Department of Financial and Management Engineering.

Tsang, S.-H., 2018. *Review: SSD — Single Shot Detector (Object Detection)*, s.l.: s.n.

Tsung-Yi, L. et al., 2015. [Online]
Available at: <https://arxiv.org/pdf/1405.0312.pdf>
[Accessed 12 August 2021].

Vignesh, S., 2020. *YOLO: You Only Look Once.*, s.l.: s.n.

Vinh, T. Q., 2020. Efficient Foreign Object Detection between PSDs and Metro Doors via Deep Neural Networks. *YUAN DAI, WEIMING LIU, HAIYU LI AND LAN LIU*, 06 March, p. 12.

Wei Liu, D. A. D. E. C. S. S. R. C.-Y. F. A. C. B., 2016. *SSD: Single Shot MultiBox Detector*. s.l., s.n.

Xue, Y., 2020. *SJTU Small Traffic Light Dataset (S2TLD)*. [Online]
Available at: <https://github.com/Thinklab-SJTU/S2TLD>

Yang, K. et al., 2021. *An Update to the ImageNet Website and Dataset*. [Online]
Available at: <https://www.image-net.org/update-mar-11-2021.php>

Yu, F., 2021. *A Diverse Driving Dataset for Heterogeneous Multitask Learning*. [Online]
Available at: <https://www.bdd100k.com/>

Zou, Z., Shi, Z., Guo, Y. & Ye, J., 2019. *Cornell University*. [Online]
Available at: <https://arxiv.org/pdf/1905.05055.pdf>
[Accessed 20 July 2021].

Appendix A. Learning in MLP networks

A.1.1 Preamble

This appendix presents every simple example of the about analysis. Its purpose is a perception of understanding the networks. This is a short but substantial tutorial that illustrates how backpropagation works in Multilayer Perception (MLP) Neural Network training. The material is taken from several scientific articles (see also references at the end of this document). We also use a practical example from the Coursera machine learning course/week 5 to explain the logic behind NN training.

Furthermore, in order to get hands-on validation of all the relationships that support NN training (and are derived below), Appendix A.2 presents and derives these relationships for a much simpler (theoretical) example.

The tutorial is structured as follows:

1. Outline of the NN structure
2. The feedforward operation during training
3. Cost function for NN training
4. Gradients of the cost function
5. Backpropagation
6. The training algorithm

A.1.2 The MLP architecture

Figure A.1 represents the NN structure in the above practical of Coursera (Machine Learning, Week 5 programming exercise). In this example, the input is a picture (image) of a handwritten digit (0 to 9) of size 20x20 pixels and the output is the number represented in the picture. The Figure A.1 presents a shallow network, however used to better understand this analysis of neural networks.

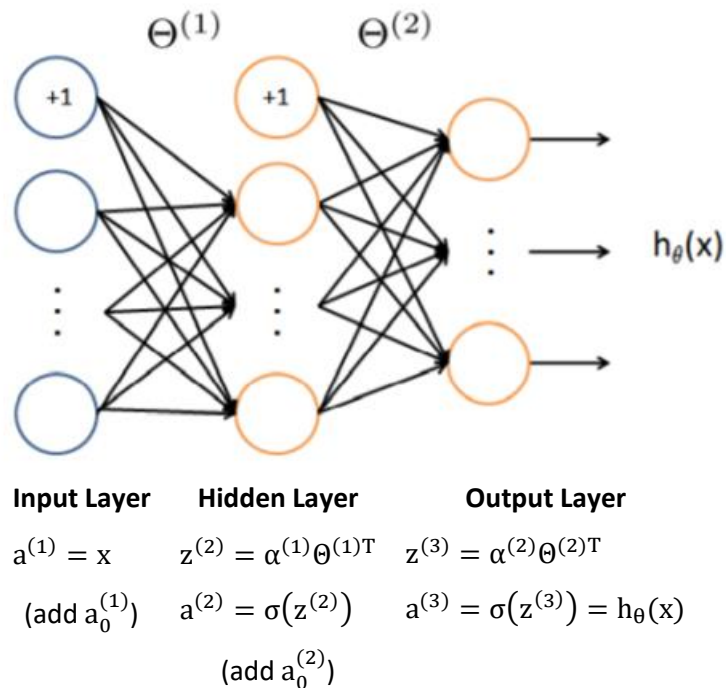


Figure A. 1 Neural network model of the example

The neural network has 3 layers:

- Input_layer_size = 400 (20x20 input pixel values of digit image) - We increase the input layer by one element $a_0^{(1)}$ in order for the matrix multiplication that provides the output of the first layer to contain the addition of the bias. Thus, the input layer size becomes 401.
- Hidden_layer_size = 25 (hidden units)
- Output layer size = 10 (output units) – The output vector contains the value of 1 in the appropriate vector element (that corresponds to the digit value) and the value 0 in all other vector elements
- Number of labels $K = 10$ (labels from 1 to 10)

Dimensions of the Θ matrices (weighting matrices) z and a (for the image above) :

- $\Theta^{(1)}$ is of size (25, 401), (the first column contains the bias elements)
- $a^{(1)}$ is of size (1,401)
- $z^{(2)}$ is of size (1,25)
- $a^{(2)}$ is of size (1,26) (be adding the bias)
- $\Theta^{(2)}$ is of size (10, 26) (the first column contains the bias elements)

- $z^{(3)}$ is of size (1,10)
- $a^{(3)}$ is of size (1,10)
- $h_{\theta}(x) = a^{(3)}$ of size (1,10) again

A.1.3 The feedforward operation during training

Consider the following training set:

- 5000 training samples (training set of 5000 images of handwritten digits, each comprising 20x20 pixels)
- The 20 by 20 grid of pixels is “unrolled” into a 400-dimensional row vector
- x is a matrix of 5000 number images of dimension (5000, 400) each image occupies a row of the matrix
- y is a matrix, each row of which represents the actual value of the corresponding sample digit. The dimension is (5000, 10) and the elements of a row are all 0 except of the element corresponding to the actual value, which is equal to 1.

The training data will be loaded into the variables x and y (by the ex4.m script in our example).

In this case (for the entire training set):

- $a^{(1)} = x$, of size (5000, 400+1)
- $z^{(2)} = \alpha^{(1)} \theta^{(1)T}$, of size (5000, 25)
- $a^{(2)} = \sigma(z^{(2)})$, of size (5000, 25+1)
- $z^{(3)} = \alpha^{(2)} \theta^{(2)T}$, of size (5000,10)

Note: *We exclude the first row from $\theta^{(2)T}$ and $\theta^{(1)T}$, and the bias units are not included when the exercise run in Octave for reduction reasons. So $a^{(2)}$ is a (5000,25) matrix and $\theta^{(2)T}$ is a (25,10) matrix and we can multiply them to find $z^{(3)}$. We are using the same mathematics to compute $a^{(1)}$ and $\theta^{(1)T}$.

$$a^{(3)} = \sigma(z^{(3)}) = h_{\theta}(x), \text{ of size (5000, 10)}$$

Note: ** σ is the Sigmoid function with a range values between (0,1).

A.1.4 The cost function for NN training

Let us start with the case of a NN with a single output

In this case, the single output NN classifies whether the input x belongs to a single class (value = 1) or not (value = 0).

In this case, let the output for the i -th training sample be $h_{\theta}(x^{(i)})$ and the true answer for that sample be $y^{(i)}$. Then the cost function will represent the sum of the errors, that is, the difference between the predicted value and the real (labeled) value.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) \quad (\text{A.1})$$

Where e.g. $m = 5000$ is the number of training samples.

Our goal is to minimize the cost function by finding $\min J(\theta)$. Note that the Sigmoid function is a “non-convex” function which means that there are multiple local minimums. So it’s not guaranteed to converge (find) the global minimum. What we need is a “convex” function in order for the gradient descent algorithm to find the global minimum (minimize $J(\theta)$). In order to do that we use the following *log* function.

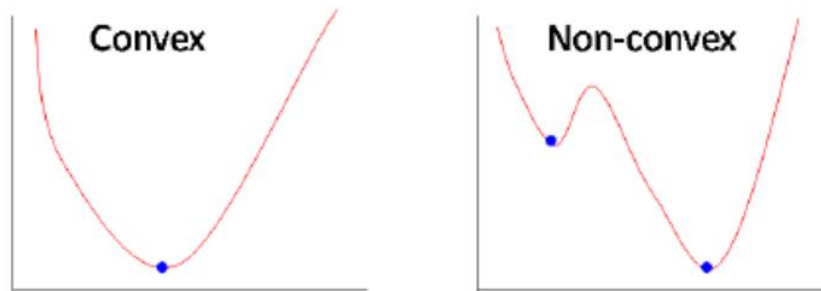


Figure A. 2 Cost function (convex&non-convex)

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -y \log(h_{\theta}(x)) & \text{if } y = 1 \\ -(1 - y) \log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases} \quad (\text{A.2})$$

Note that

- if $y = h_{\theta}(x) = 1$, then the cost is zero, since $\log(1) = 0$
- if $y = 1$ and $h_{\theta}(x) = 0$, then the cost is ∞ , since $\log(0) = -\infty$

Similarly

- if $y = h_\theta(x) = 0$, then the cost is zero, since $\log(1) = 0$
- if $y = 0$ and $h_\theta(x) = 1$, then the cost is ∞ , since $\log(0) = -\infty$

Since y (labeled value) is either 0 or 1 we can write the cost function in one equation.

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x)) \quad (\text{A.3})$$

For the m training samples, the cost function for this single output NN becomes

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \quad (\text{A.4})$$

To illustrate that the cost function is a convex function we plot a simple example using python. Consider that the input $y^{(i)} = 1$. Then

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log(h_\theta(x^{(i)})) \quad (\text{A.5})$$

As we already know $a^{(3)} = \sigma(z^{(3)}) = h_\theta(x)$. Let $h_\theta(x^{(i)}) \in [0, 1]$ increasing from zero to 1 by 0.1 in every iteration of the numerical example. Then, $J(\theta)$ has the convex form of Fig. A.2, which is hardly surprising given its logarithmic nature. This applies only to logistic regression i.e. a neural network with no hidden layers.

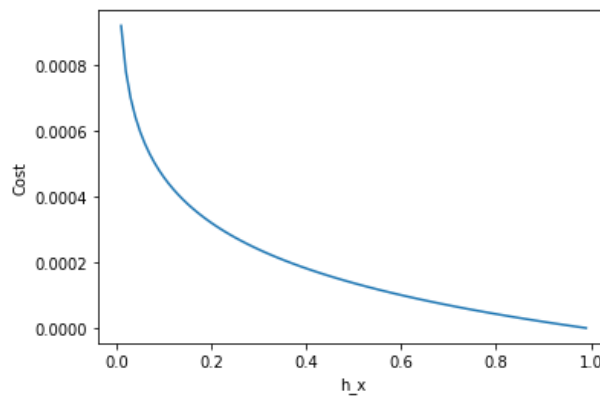
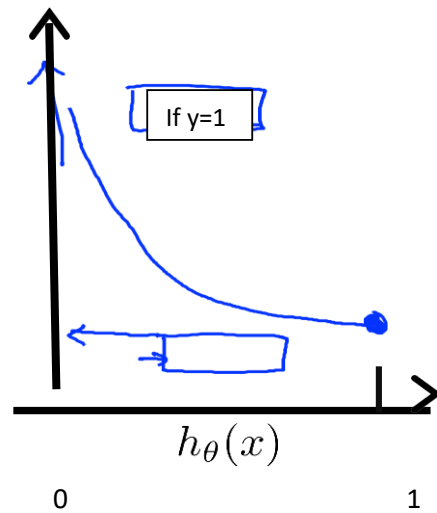


Figure A. 3 The convex shape of a simple instance of the cost function

This is the same result presented on the machine learning lectures (week 5, Coursera) – see below.



Cost = 0 if $y = 1, h_{\theta}(x) = 1$

But as $h_{\theta}(x) \rightarrow 0$

$Cost \rightarrow \infty$

Captures intuition that if $h_{\theta}(x) = 0$, (predict $P(y = 1|x; \theta) = 0$), but $y = 1$, we'll penalize learning algorithm by a very large cost.

Figure A. 4 Convex between $h_{\theta}(x)$ and Cost(J) if $y=1$.

The cost function of Eq. (A.4) does not include regularization. Note that in our example, the number of elements of the θ matrices is over 10,000 and the number of training samples is just 5,000, which means that theoretically we have more than adequate parameters to obtain a value of 0 for $J(\theta)$ (consider a system with 10,000 unknowns, the elements of θ and 5,000 equations) that may result to overfitting. In order to address this overfitting risk, we reduce the magnitude/values of θ (making many of them to be 0) by introducing a penalty term as below.

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2 \quad (\text{A.6})$$

Equation (A.6) represents the regularized cost function.

Let's move now to our example that has multiple outputs

If we generalize the above for multiple NN output nodes (multiclass classification) what we get is:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m=5000} \sum_{k=1}^{K=10} [-y_k^{(i)} \log(h_\theta(x^{(i)})_k) - (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k)] + \frac{\lambda}{2m} \left[\sum_{j=1}^{25} \sum_{n=1}^{400} (\theta_{j,n}^{(1)})^2 + \sum_{j=1}^{10} \sum_{n=1}^{25} (\theta_{j,n}^{(2)})^2 \right] \quad (A.7)$$

where in our example $K = 10$ is the number of outputs/labels and $m = 5000$ is the number of training samples, λ is the regularization factor, $h_\theta(x) \in \mathbb{R}^K$ and $(h_\theta(x^{(i)}))_k$ is the value of the k -th output for the i -th training sample.

A.1.5 Gradient of the cost function (without regularization)

Gradient descent is an optimization algorithm used to minimize our cost function. In general, it is used to find the values of the parameters that optimize a non-linear objective function.

In machine learning, specifically, we use gradient descent to determine the parameters of our NN model during training. Note that the non-linear optimization problem we deal with is relatively straightforward, since it does not involve any constraints (just the objective function). We can think of the gradient as the slope of the function. The higher the gradient, the steeper the slope and the faster a model can learn (determine the appropriate values for its parameters). If the slope is zero, the model stops learning.

Given a training set, the cost function $J(\theta)$ depends strictly on the values of NN parameters, the weights $\theta^{(1)}$ and $\theta^{(2)}$. Thus, in our example training of the NN is the process of determining the values of θ that drive the value of the cost function of Section A.1.4 to its minimum. We should start the process by setting initial values of the parameters, and gradient descent will iteratively adjust these values to minimize the cost-function based on the following relationship:

$$\theta_{j+1} = \theta_j - \alpha \nabla J(\theta_j) \quad (A.8)$$

where: θ_{j+1} is the value of the next iteration

θ_j is the value of the current iteration

α is the step along the gradient

$\nabla J(\theta_j)$ is the gradient, i.e., the vector of partial derivatives of J with respect to each parameter of θ at point θ_j . It is simply the direction of the steepest slope of the function at this point.

Concerning step α (or learning rate), it must be set to an appropriate value, which is neither too low nor too high. This is important because if the steps are excessively long, the algorithm may overshoot the minimum. If the learning rate is too low, the process may take excessive time to reach the local minimum, or it may never reach it due to excessively slow convergence.

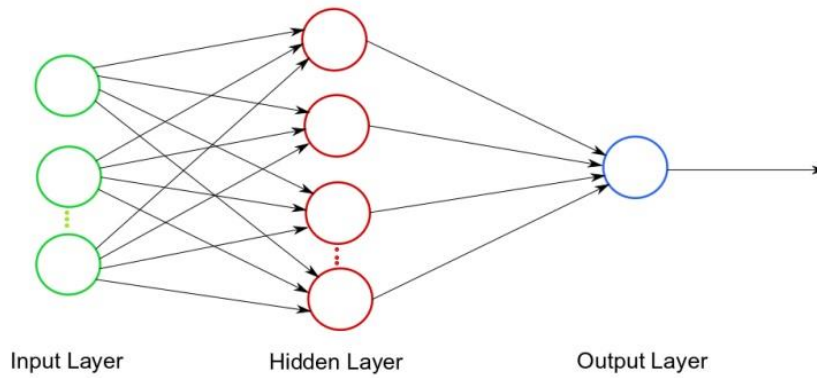
The values of θ that correspond to the minimum are the final values of the NN parameters to be used thereafter and the NN has been trained. Thus, in order to determine the minimum of the cost function and the values of θ that correspond to this minimum, the most computationally intensive task is to determine $\nabla J(\theta_j)$ at each iteration j .

This is achieved by backpropagation, which uses the output of NN (h), compares it to the real value (y) and derives the error (δ). The errors for each layer can be used to calculate the partial derivatives. In our example, starting from the final layer $L = 3$, backpropagation attempts to define the error value $\delta_k^{(L)}$ where k is the node and L is the layer.

In order to define the gradient of the cost function with respect to the parameters θ , we will start from the single output NN and we will generalize to the K output NN. In both cases we will start from the last layer.

A.1.5.1 Single output NN

Consider the following network



$$\begin{array}{lll}
 a^{(1)} = x & z^{(2)} = \alpha^{(1)} \theta^{(1)T} & z^{(3)} = \alpha^{(2)} \theta^{(2)T} \\
 (\text{add } a_0^{(1)}) & a^{(2)} = \sigma(z^{(2)}) & a^{(3)} = \sigma(z^{(3)}) = h_{\theta}(x) \\
 & (\text{add } a_0^{(2)}) &
 \end{array}$$

Dimensions of the θ matrices (weighting matrices) z and a (for the image above):

- $\theta^{(1)}$ is of size (25, 401), (the first column contains the bias elements).
- $a^{(1)}$ is of size (1,401)
- $z^{(2)}$ is of size (1,25)
- $a^{(2)}$ is of size (1,26)(be adding bias)
- $\theta^{(2)}$ is of size (1, 26) (the first column contains the bias elements)
- $z^{(3)}$ is of size (1,1)
- $a^{(3)}$ is of size (1,1)
- $h_{\theta}(x) = a^{(3)}$ is of size (1,1)

In this case (for the entire training set):

- $a^{(1)} = x$, of size (5000, 400+1)
- $z^{(2)} = \alpha^{(1)} \theta^{(1)T}$, of size (5000, 25)
- $a^{(2)} = \sigma(z^{(2)})$, of size (5000, 25+1)
- $z^{(3)} = \alpha^{(2)} \theta^{(2)T}$, of size (5000,1)

Note: The first row of the $\theta^{(2)T}$ and $\theta^{(1)T}$ matrices contain the bias elements. We can exclude this row without affecting our results. Additionally, this will reduce the data and make them easier to compute. As a result, $a^{(2)}$ is a (5000,25) matrix, $\theta^{(2)T}$ is a

(25,1) matrix and are multiplied to compute $z^{(3)}$. We are using the similar operations to compute $z^{(2)}$ from $a^{(1)}$ and $\theta^{(1)T}$.

As we have already mentioned and analyzed above, the cost function for a single output NN is:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

We can compute the gradient $\nabla J(\theta)$ from the chain rule

$$\nabla J(\theta) = \frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J}{\partial \alpha} \frac{\partial \alpha}{\partial z} \frac{\partial z}{\partial \theta}$$

We will now focus on the third layer to determine the partial derivatives with respect to elements $\theta_j^{(2)}$ of $\Theta^{(2)}$, $j = 1, \dots, 25$

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_j^{(2)}} &= -\frac{1}{m} \sum_{i=1}^m \frac{\partial [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]}{\partial \theta_j^{(2)}} \\ &= \frac{1}{m} \sum_{i=1}^m \frac{d[-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]}{da^{(3)(i)}} \frac{da^{(3)(i)}}{dz^{(3)(i)}} \frac{\partial z^{(3)(i)}}{\partial \theta_j^{(2)}} \\ &= \frac{1}{m} \sum_{i=1}^m \frac{d\hat{f}_i}{da^{(3)(i)}} \frac{da^{(3)(i)}}{dz^{(3)(i)}} \frac{\partial z^{(3)(i)}}{\partial \theta_j^{(2)}} \end{aligned}$$

where the notation $\frac{d\hat{f}_i}{da^{(3)(i)}}$ represents the derivative $\frac{df}{da^{(3)}}$ evaluated using the values of the (i)-th training instance. The same notation is used for $\frac{da^{(3)(i)}}{dz^{(3)(i)}}$, and $\frac{\partial z^{(3)(i)}}{\partial \theta_j^{(2)}}$.

$$\begin{aligned}\frac{d\hat{J}}{da^{(3)}} &= \frac{d\hat{J}}{dh_{\theta}(x)} = \frac{d}{dh_{\theta}(x)} [(-y\log(h_{\theta}(x)) - (1-y)\log(1-h_{\theta}(x)))] \\ &= -\frac{y}{h_{\theta}(x)} + \frac{1-y}{1-h_{\theta}(x)} = -\frac{y}{a^{(3)}} + \frac{1-y}{1-a^{(3)}}\end{aligned}$$

Thus

$$\frac{d\hat{J}_i}{da^{(3)(i)}} = -\frac{y^{(i)}}{a^{(3)(i)}} + \frac{1-y^{(i)}}{1-a^{(3)(i)}} \quad (\text{A.9})$$

$$\begin{aligned}\frac{da^{(3)}}{dz^{(3)}} &= \frac{d}{dz^{(3)}} \frac{1}{1+e^{-z^{(3)}}} = \frac{d}{dz^{(3)}} \left(1+e^{-z^{(3)}}\right)^{-1} = -\left(1+e^{-z^{(3)}}\right)^{-2} \left(-e^{-z^{(3)}}\right) \\ &= \frac{e^{-z^{(3)}}}{\left(1+e^{-z^{(3)}}\right)^2} = \frac{1}{1+e^{-z^{(3)}}} \left(1 - \frac{1}{1+e^{-z^{(3)}}}\right) = \sigma(z^{(3)}) \left(1 - \sigma(z^{(3)})\right) \\ &= a^{(3)}(1-a^{(3)})\end{aligned} \quad (\text{A.10})$$

$$\text{So, } \frac{da^{(3)(i)}}{dz^{(3)(i)}} = a^{(3)(i)}(1-a^{(3)(i)})$$

$$\frac{\partial z^{(3)}}{\partial \theta_j^{(2)}} = \frac{\partial}{\partial \theta_j^{(2)}} (a^{(2)} \Theta^{(2)T}) = a_j^{(2)} \rightarrow \frac{\partial z^{(3)(i)}}{\partial \theta_j^{(2)}} = a_j^{(2)(i)} \quad (\text{A.11})$$

Combining the above for the output layer:

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_j^{(2)}} &= \frac{1}{m} \sum_{i=1}^m \left(-\frac{y^{(i)}}{a^{(3)(i)}} + \frac{1-y^{(i)}}{1-a^{(3)(i)}} \right) a^{(3)(i)} (1-a^{(3)(i)}) a_j^{(2)(i)} \\ &= \frac{1}{m} \sum_{i=1}^m (a^{(3)(i)} - y^{(i)}) a_j^{(2)(i)} \\ \frac{\partial J(\theta)}{\partial \theta_j^{(2)}} &= \frac{1}{m} \sum_{i=1}^m \delta^{(3)(i)} a_j^{(2)(i)}\end{aligned} \quad (\text{A.12})$$

where $\delta^{(3)(i)} = a^{(3)(i)} - y^{(i)} = h_{\theta}(x^{(i)}) - y^{(i)}$ is a (1,1) vector of the error for training instance i . Moreover, we can combine the results of Eq. (12) for all $j = 1, \dots, 25$ to obtain $\frac{\partial J(\theta)}{\partial \theta^{(2)}}$, an (1,25) vector.

A.1.5.2 Multiple output NN

We will now generalize to the K output NN. Again, from the chain rule

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J}{\partial \alpha} \frac{\partial \alpha}{\partial z} \frac{\partial z}{\partial \theta}$$

Let's focus on the third layer as before

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} &= \frac{1}{m} \sum_{i=1}^m \frac{\partial [-y_k^{(i)} \log(h_{\theta}(x^{(i)})_k) - (1 - y_k^{(i)}) \log(1 - h_{\theta}(x^{(i)})_k)]}{\partial \theta_{kj}^{(2)}} \\ &= \frac{1}{m} \sum_{i=1}^m \frac{d[-y_k^{(i)} \log(h_{\theta}(x^{(i)})_k) - (1 - y_k^{(i)}) \log(1 - h_{\theta}(x^{(i)})_k)]}{da_k^{(3)(i)}} \frac{da_k^{(3)(i)}}{dz_k^{(3)(i)}} \frac{\partial z_k^{(3)(i)}}{\partial \theta_{kj}^{(2)}} \quad (\text{A.13}) \\ &= \frac{1}{m} \sum_{i=1}^m \frac{d\hat{f}_{ki}}{da_k^{(3)(i)}} \frac{da_k^{(3)(i)}}{dz_k^{(3)(i)}} \frac{\partial z_k^{(3)(i)}}{\partial \theta_{kj}^{(2)}} \end{aligned}$$

where again the notation $\frac{d\hat{f}_{ki}}{da_k^{(3)(i)}}$ represents the derivative $\frac{d\hat{f}_k}{da_k^{(3)}}$ evaluated using the

values of the (i)-th training instance. The same notation is used for $\frac{da_k^{(3)(i)}}{dz_k^{(3)(i)}}$, and $\frac{\partial z_k^{(3)(i)}}{\partial \theta_{kj}^{(2)}}$

In the first row of Eq. (A.13) the summation $\sum_{k=1}^K$ of the cost function $J(\theta)$ does not appear, since $\frac{\partial [-y_n^{(i)} \log(h_{\theta}(x^{(i)})_n) - (1 - y_n^{(i)}) \log(1 - h_{\theta}(x^{(i)})_n)]}{\partial \theta_{kj}^{(2)}}$ is zero if $n \neq k$.

$$\begin{aligned} \frac{d\hat{f}_k}{da_k^{(3)}} &= \frac{d\hat{f}_k}{dh_{\theta}(x)_k} = \frac{d}{dh_{\theta}(x)_k} [-y_k \log(h_{\theta}(x)_k) - (1 - y_k) \log(1 - h_{\theta}(x)_k)] \\ &= -\frac{y_k}{h_{\theta}(x)_k} + \frac{1 - y_k}{1 - h_{\theta}(x)_k} = -\frac{y_k}{a_k^{(3)}} + \frac{1 - y_k}{1 - a_k^{(3)}} \end{aligned}$$

Thus

$$\frac{d\hat{f}_{ki}}{da_k^{(3)(i)}} = -\frac{y_k^{(i)}}{a_k^{(3)(i)}} + \frac{1 - y_k^{(i)}}{1 - a_k^{(3)(i)}} \quad (\text{A.14})$$

From Eq. (A.10)

$$\frac{da_k^{(3)(i)}}{dz_k^{(3)(i)}} = a_k^{(3)(i)} (1 - a_k^{(3)(i)}) \quad (\text{A.15})$$

Furthermore

$$\frac{\partial z_k^{(3)}}{\partial \theta_{kj}^{(2)}} = \frac{\partial}{\partial \theta_{kj}^{(2)}} (a^{(2)} \theta^{(2)T}) = a_{kj}^{(2)} \rightarrow \frac{\partial z_k^{(3)(i)}}{\partial \theta_{kj}^{(2)}} = a_j^{(2)(i)} \quad (\text{A.16})$$

Combining the above for the output layer:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} &= \frac{1}{m} \sum_{i=1}^m \left(-\frac{y_k^{(i)}}{a_k^{(3)(i)}} + \frac{1 - y_k^{(i)}}{1 - a_k^{(3)(i)}} \right) a_k^{(3)(i)} (1 - a_k^{(3)(i)}) a_j^{(2)(i)} \\ &= \frac{1}{m} \sum_{i=1}^m (a_k^{(3)(i)} - y_k^{(i)}) a_j^{(2)(i)} \end{aligned}$$

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(3)(i)} a_j^{(2)(i)} \quad (\text{A.17})$$

with $\delta_k^{(3)(i)} = a_k^{(3)(i)} - y_k^{(i)} = h_{\theta}(x^{(i)})_k - y_k^{(i)}$ is the error of output $k = 1, \dots, 10$ for training instance i . Moreover, we can combine the results of Eq. (A.17) for all $k = 1, \dots, 10$, $j = 1, \dots, 25$ to obtain $\frac{\partial J(\theta)}{\partial \theta^{(2)}}$, an (1,250) vector.

A.1.5.3 Gradient for all NN layers

Following the same process as in Sections A.5.1 and A.5.2, one can obtain

For the single output NN and for the output layer L

$$\frac{\partial J(\theta)}{\partial \theta_j^{(L-1)}} = \frac{1}{m} \sum_{i=1}^m \delta^{(L)(i)} a_j^{(L-1)(i)} \quad (\text{A.18})$$

For all the other layers (l) of the single output NN and for all layers ($l = 1, \dots, L$) of the multiple output NN

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(l)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(l+1)(i)} a_j^{(l)(i)} \quad (\text{A.19})$$

Depending on the case, using Eq. (A.18) or (A.19) we may compute the entire gradient vector $\nabla J(\theta)$ from the outputs of the NN $a_j^{(l)(i)}$ (from forward propagation) provided we know the errors $\delta_k^{(l+1)(i)}$. The errors are known only for the last layer $l = L$, i.e. $\delta_k^{(L)(i)} = a_k^{(3)(i)} - y_k^{(i)}$, again from forward propagation. For $l < L$, the errors $\delta_k^{(l)(i)}$ are obtained from back propagation as discussed below.

A.1.6 Backpropagation

Let's now compute the errors δ involved in Eq. (A.18) or (A.19) that provide the gradients of the cost function for label k and layer l . For simplicity, we will again use our example with the three layers.

A.1.6.1 Errors of layer $L = 3$

As we have discussed above, the error for the final layer (in our example layer 3) for output $k = 1, \dots, K = 10$ is determined using forward propagation by

$$\delta_k^{(3)} = a_k^{(3)} - y_k$$

or for each training instance i

$$\delta_k^{(3)(i)} = a_k^{(3)(i)} - y_k^{(i)} \quad (\text{A.20})$$

and may be obtained directly from forward propagation by subtracting the actual value $y_k^{(i)}$ of output k of instance i from the NN output $a_k^{(3)(i)}$ of label k of instance i .

A.1.6.2 Errors of layer $l = 2$

Consider the single output NN. We use the following notation:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] = \frac{1}{m} \sum_{i=1}^m \hat{f}_i$$

Without proof the following holds:

$$\delta_j^{(2)(i)} = \frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}} \quad (\text{A.21})$$

Where $\frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}}$ is the value of the derivative of $\frac{\partial \hat{J}(\theta)}{\partial z_j^{(2)}}$ for training instance i. We will evaluate this derivative for instance i.

$$\begin{aligned} \frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}} &= \frac{\partial [-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]}{\partial z_j^{(2)(i)}} \\ &= \frac{d[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]}{da^{(3)}} \frac{da^{(3)(i)}}{dz^{(3)(i)}} \frac{\partial z^{(3)(i)}}{\partial a_j^{(2)(i)}} \frac{da_j^{(2)(i)}}{dz_j^{(2)(i)}} \end{aligned} \quad (\text{A.22})$$

From Eq. (A.9)

$$\frac{d[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]}{da^{(3)(i)}} = -\frac{y^{(i)}}{a^{(3)(i)}} + \frac{1 - y^{(i)}}{1 - a^{(3)(i)}}$$

From Eq. (A.10)

$$\begin{aligned} \frac{da^{(3)(i)}}{dz^{(3)(i)}} &= a^{(3)(i)}(1 - a^{(3)(i)}) \\ \frac{\partial z^{(3)(i)}}{\partial a_j^{(2)(i)}} &= \frac{\partial [\alpha^{(2)(i)} \Theta^{(2)T}]}{\partial a_j^{(2)(i)}} = \theta_j^{(2)} \end{aligned}$$

with $\theta_j^{(2)}$ the j – th element of $\Theta^{(2)}$. In the single output NN, $\Theta^{(2)}$ is a row vector, e.g. (1,25).

Finally

$$\frac{da_j^{(2)(i)}}{dz_j^{(2)(i)}} = a_j^{(2)(i)}(1 - a_j^{(2)(i)})$$

Combining the above we obtain

$$\begin{aligned} \frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}} &= \left(-\frac{y^{(i)}}{a^{(3)(i)}} + \frac{1 - y^{(i)}}{1 - a^{(3)(i)}} \right) a^{(3)(i)}(1 - a^{(3)(i)}) \theta_j^{(2)} a_j^{(2)(i)}(1 - a_j^{(2)(i)}) \\ &= \delta^{(3)(i)} \theta_j^{(2)} [a_j^{(2)(i)}(1 - a_j^{(2)(i)})] \end{aligned}$$

That is,

$$\frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}} = \delta^{(3)(i)} \theta_j^{(2)} a_j^{(2)(i)} (1 - a_j^{(2)(i)}) = \delta_j^{(2)(i)} \quad (\text{A.23})$$

The last equality coming from Eq. (A.21). Now if we consider

$$\frac{\partial \hat{J}(\theta)_i}{\partial \mathbf{z}^{(2)(i)}} = \left(\frac{\partial \hat{J}(\theta)_i}{\partial z_1^{(2)(i)}}, \dots, \frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}}, \dots \right)$$

then

$$\frac{\partial \hat{J}(\theta)_i}{\partial \mathbf{z}^{(2)(i)}} = (\delta^{(3)(i)} \boldsymbol{\theta}^{(2)}) .* (\mathbf{a}^{(2)(i)} .* (1 - \alpha^{(2)(i)})) \quad (\text{A.24})$$

where the symbol $(.*)$ represents the element-wise multiplication of two matrices (vectors in this particular case).

Then from Eq. (A.21)

$$\delta^{(2)(i)} = \frac{\partial \hat{J}(\theta)_i}{\partial z_j^{(2)(i)}} = (\delta^{(3)(i)} \boldsymbol{\theta}^{(2)}) .* (\mathbf{a}^{(2)(i)} .* (1 - \alpha^{(2)(i)})) \quad (\text{A.25})$$

Now consider the multiple output NN

With similar arguments we obtain the following equation:

$$\delta^{(2)(i)} = (\delta^{(3)(i)} \boldsymbol{\theta}^{(2)}) .* (\mathbf{a}^{(2)(i)} .* (1 - \alpha^{(2)(i)})) \quad (\text{A.26})$$

A.1.6.3 Errors of layer l

Now let us generalize to error $\delta^{(l)}$ for layer $l < L$ in terms of the error $\delta^{(l+1)}$ of layer $l + 1$

$$\delta^{(l)(i)} = (\delta^{(l+1)(i)} \boldsymbol{\theta}^{(l)}) .* (\alpha^{(l)(i)} .* (1 - \alpha^{(l)(i)})) \quad (\text{A.27})$$

or

$$(\text{A.28})$$

$$\delta^{(l)} = (\delta^{(l+1)} \theta^{(l)}) .* (\alpha^{(l)} .* (1 - \alpha^{(l)}))$$

This last equation may be considered equivalent to Eq. (A.27) if $\delta^{(l)} = \frac{1}{m} \sum_{i=1}^m \delta^{(l)(i)}$.

Equation (A.28) moves the error backwards through the activation function of layer l , giving us the error $\delta^{(l)}$ as the weighted sum of error $\delta^{(l+1)}$ of layer $l + 1$. The initial error of the last layer is, of course, obtained directly by subtracting the actual value of the output from the estimated value of the output (NN output).

From Eq. (A.27) we may get the δ terms of each layer. Then, we use them in Eq. (A.19) to obtain the partial derivative of the error function J with respect to individual parameters of the NN and thus compute the gradient ∇J of Eq. (A.8), which is used in the related step of the gradient descent. This process is repeated for each step, since the θ values are updated and so are the terms $a_j^{(l)(i)}$ of forward propagation (evaluated with the new θ) and $\delta_k^{(l+1)(i)}$ of backpropagation. The process minimizes J with respect to the NN parameters θ and trains the NN by obtaining the optimal values of θ .

A.1.7 A theoretical validation example

In order to obtain a hands-on understanding of the forward and backpropagation relationships used in NN training, as well as their proofs, Appendix A.2 presents and proves these relationships for a very simple (but theoretical) example.

A.1.8 The training algorithm

In order to put together the mathematical concepts of Sections A.1.5 (forward propagation) and A.1.6 (backpropagation), we present the following algorithm for the original example of Section 1.

Training is based on a set $[(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})]$, where $m = 5,000$, that is considered as input to the algorithm.

Step 1

Start training by initializing the values of θ_1 and θ_2 with small random numbers equal to zero or near it. Then, gradient descent will update the θ_1 and θ_2 values in an attempt to minimize the error.

Step 2

Perform forward propagation to compute $\alpha^{(l)(i)}$ for layers ($l = 2,3$) for the training sample i ($i = 1, \dots, 5000$). Forward propagation uses the following equations:

Input to hidden layer : $a^{(1)(i)} = x^{(i)}$, of size (1,400+1)

$$z^{(2)(i)} = \alpha^{(1)(i)} \theta^{(1)(i)T}, \text{ of size } (1,25)$$

Hidden to output layer : $a^{(2)(i)} = \sigma(z^{(2)(i)})$, of size (1,25+1)

$$z^{(3)(i)} = \alpha^{(2)(i)} \theta^{(2)(i)T}, \text{ of size } (1,10)$$

$$a^{(3)(i)} = \sigma(z^{(3)(i)}) = h_{\theta}(x), \text{ of size } (1,10)$$

Forward propagation provides $a_k^{3(i)}$, $k = 1, \dots, 10$, $i = 1, \dots, 5000$ to be used in Eq. (A.19) and (A.20), as well as $a_j^{2(i)}$, $j = 1, \dots, 25$ to be used in Eq. (A.27)

Step 3

Compute:

$$\delta_k^{(3)(i)} = a_k^{(3)(i)} - y_k^{(i)}, k = 1, \dots, 10, i = 1, \dots, 5000$$

$$\delta^{2(i)} = (\delta^{(2)(i)} \theta^{(2)}) .* (\alpha^{(2)(i)} .* (1 - \alpha^{(2)(i)})), i = 1, \dots, 5000$$

with $\delta^{2(i)} = (\delta_1^{(2)(i)}, \dots, \delta_j^{(2)(i)}, \dots, \delta_{25}^{(2)(i)})$, an (1, 25) vector

Step 4

Using the results of Steps 2 and 3, compute

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(3)(i)} a_j^{(2)(i)}, k = 1, \dots, 10, \quad j = 1, \dots, 25$$

and

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(1)}} = \frac{1}{m} \sum_{i=1}^m \delta_k^{(2)(i)} a_j^{(1)(i)}, \quad k = 1, \dots, 25, \quad j = 1, \dots, 400$$

This is without regularization. We can easily add regularization.

Step 5

Having obtained $\nabla J(\theta_j) = \frac{\partial J(\theta)}{\partial \theta} |_{\theta = \theta_j}$ an $(1, 10250)$ vector, then update θ using

$$\theta_{j+1} = \theta_j - a \nabla J(\theta_j)$$

With a the chosen step.

Step 6

Repeat steps 1-5 till $\|\nabla J(\theta_j)\| < \varepsilon$. Set θ equal to the values of the parameters of the last iteration. The NN has been trained.

A.2 Feedforward and backpropagation equations

Consider a very simple neural network that has 2 input nodes, 3 hidden nodes, and 2 output nodes (see Fig. A.1). The vectors and the Θ matrices (weighting matrices) involved are the following:

$$\alpha^{(1)} = [x_1 \ x_2] = [\alpha_1^{(1)} \ \alpha_2^{(1)}]$$

$$\theta^{(1)} = \begin{bmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} \\ \theta_{31}^{(1)} & \theta_{32}^{(1)} \end{bmatrix}$$

$$z^{(2)} = [z_1^{(2)} \ z_2^{(2)} \ z_3^{(2)}]$$

$$\alpha^{(2)} = [\alpha_1^{(2)} \ \alpha_2^{(2)} \ \alpha_3^{(2)}]$$

$$\theta^{(2)} = \begin{bmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} & \theta_{13}^{(2)} \\ \theta_{21}^{(2)} & \theta_{22}^{(2)} & \theta_{23}^{(2)} \end{bmatrix}$$

$$\mathbf{z}^{(3)} = \begin{bmatrix} z_1^{(3)} & z_2^{(3)} \end{bmatrix}$$

$$\boldsymbol{\alpha}^{(3)} = \begin{bmatrix} \alpha_1^{(3)} & \alpha_2^{(3)} \end{bmatrix}$$

A.2.1 Forward propagation

The forward propagation relationships are as follows:

Layer 2

$$\mathbf{z}^{(2)} = \boldsymbol{\alpha}^{(1)} \boldsymbol{\theta}^{(1)T} \rightarrow \mathbf{z}^{(2)} = \begin{bmatrix} \alpha_1^{(1)} & \alpha_2^{(1)} \end{bmatrix} \begin{bmatrix} \theta_{11}^{(1)} & \theta_{21}^{(1)} & \theta_{31}^{(1)} \\ \theta_{12}^{(1)} & \theta_{22}^{(1)} & \theta_{32}^{(1)} \end{bmatrix}$$

or

(A.29)

$$z_1^{(2)} = \alpha_1^{(1)} \theta_{11}^{(1)} + \alpha_2^{(1)} \theta_{12}^{(1)}$$

$$z_2^{(2)} = \alpha_1^{(1)} \theta_{21}^{(1)} + \alpha_2^{(1)} \theta_{22}^{(1)}$$

$$z_3^{(2)} = \alpha_1^{(1)} \theta_{31}^{(1)} + \alpha_2^{(1)} \theta_{32}^{(1)}$$

Then

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$$

or

$$\alpha_1^{(2)} = \sigma(z_1^{(2)}) = \frac{1}{1 + e^{-z_1^{(2)}}} \quad (A.30)$$

$$\alpha_2^{(2)} = \sigma(z_2^{(2)}) = \frac{1}{1 + e^{-z_2^{(2)}}}$$

$$\alpha_3^{(2)} = \sigma(z_3^{(2)}) = \frac{1}{1 + e^{-z_3^{(2)}}}$$

Layer 3

$$z^{(3)} = \alpha^{(2)} \theta^{(2)T} \Rightarrow z^{(3)} = [\alpha_1^{(2)} \quad \alpha_2^{(2)} \quad \alpha_3^{(2)}] \begin{bmatrix} \theta_{11}^{(2)} & \theta_{21}^{(2)} \\ \theta_{12}^{(2)} & \theta_{22}^{(2)} \\ \theta_{13}^{(2)} & \theta_{23}^{(2)} \end{bmatrix} \quad (\text{A.31})$$

or

$$z_1^{(3)} = \alpha_1^{(2)} \theta_{11}^{(2)} + \alpha_2^{(2)} \theta_{12}^{(2)} + \alpha_3^{(2)} \theta_{13}^{(2)}$$

$$z_2^{(3)} = \alpha_1^{(2)} \theta_{21}^{(2)} + \alpha_2^{(2)} \theta_{22}^{(2)} + \alpha_3^{(2)} \theta_{23}^{(2)}$$

Then

$$a^{(3)} = \sigma(z^{(3)}) = h_{\theta}(x)$$

$$\alpha_1^{(3)} = \sigma(z_1^{(3)}) = \frac{1}{1 + e^{-z_1^{(3)}}} \quad (\text{A.32})$$

$$\alpha_2^{(3)} = \sigma(z_2^{(3)}) = \frac{1}{1 + e^{-z_2^{(3)}}}$$

Note that the vector of the NN parameters is θ and contains 6+6=12 parameters $\theta_{kj}^{(l)}$

$$\theta^{(1)} = \begin{bmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} \\ \theta_{31}^{(1)} & \theta_{32}^{(1)} \end{bmatrix} \quad \theta^{(2)} = \begin{bmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} & \theta_{13}^{(2)} \\ \theta_{21}^{(2)} & \theta_{22}^{(2)} & \theta_{23}^{(2)} \end{bmatrix}$$

A.2.2 The cost function

Consider now that the training set consists of two training samples ($i = 1, 2$), which of course is unrealistic, but it is simple enough for the theoretical example. Thus the training set is $\{(x_1^{(1)}, x_2^{(1)}), (y_1^{(1)}, y_2^{(1)}); (x_1^{(2)}, x_2^{(2)}), (y_1^{(2)}, y_2^{(2)})\}$

If we fully write our cost function with the summation we would get:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^2 \left\{ [-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] + [-y_2^{(i)} \log(\alpha_2^{(3)(i)}) - (1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)})] \right\} \quad (\text{A.33})$$

A.2.3 Partial derivatives of $J(\theta)$ with respect to the weights $\theta_{11}^{(2)}, \dots, \theta_{23}^{(2)}$ (6 parameters)

Consider

$$\begin{aligned} & \frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} \\ &= \frac{1}{2} \sum_{i=1}^2 \frac{\partial \left\{ [-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] + [-y_2^{(i)} \log(\alpha_2^{(3)(i)}) - (1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)})] \right\}}{\partial \theta_{kj}^{(2)}} \\ &= \frac{1}{2} \sum_{i=1}^2 \frac{d \left\{ [-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] + [-y_2^{(i)} \log(\alpha_2^{(3)(i)}) - (1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)})] \right\}}{d \alpha_k^{(3)(i)}} \\ & \times \frac{d \alpha_k^{(3)(i)}}{d z_k^{(3)(i)}} \frac{\partial z_k^{(3)(i)}}{\partial \theta_{kj}^{(2)}} \end{aligned} \quad (\text{A.34})$$

Let $k=1$ and $j=2$

$$\frac{d \left\{ [-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] \right\}}{d \alpha_1^{(3)(i)}} = \frac{-y_1^{(i)}}{\alpha_1^{(3)(i)}} + \frac{(1 - y_1^{(i)})}{1 - \alpha_1^{(3)(i)}} \quad (\text{A.35})$$

And the derivative of the second term of the numerator in the first equation of

Eq. (A.34) with respect to $d \alpha_1^{(3)(i)}$ is zero. Furthermore,

$$\begin{aligned}
\frac{d\alpha_1^{(3)(i)}}{dz_1^{(3)(i)}} &= \frac{-1}{\left(1 + e^{-z_1^{(3)(i)}}\right)^2} \left(-e^{-z_1^{(3)(i)}}\right) = \frac{e^{-z_1^{(3)(i)}}}{\left[1 + e^{-z_1^{(3)(i)}}\right]^2} \\
&= \frac{1}{1 + e^{-z_1^{(3)(i)}}} \left[1 - \frac{1}{1 + e^{-z_1^{(3)(i)}}}\right] = \alpha_1^{(3)(i)}(1 - \alpha_1^{(3)(i)})
\end{aligned} \tag{A.36}$$

$$\frac{\partial z_1^{(3)(i)}}{\partial \theta_{12}^{(2)}} = \alpha_2^{(2)(i)} \tag{A.37}$$

from equation for $z_1^{(3)}$ in Eq. (A.31)

Thus, substituting Eqs. (A-6) to (A-8) into (A-5) for $k=1$ and $j=2$, we obtain

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} &= \frac{1}{2} \sum_{i=1}^2 \left(\frac{-y_1^{(i)}}{\alpha_1^{(3)(i)}} + \frac{(1 - y_1^{(i)})}{1 - \alpha_1^{(3)(i)}} \right) \left[\alpha_1^{(3)(i)}(1 - \alpha_1^{(3)(i)}) \right] \alpha_2^{(2)(i)} = \\
&= \frac{1}{2} \sum_{i=1}^2 \alpha_2^{(2)(i)} \{ -y_1^{(i)}(1 - \alpha_1^{(3)(i)}) + (1 - y_1^{(i)})\alpha_1^{(3)(i)} \} \\
&= \frac{1}{2} \sum_{i=1}^2 \alpha_2^{(2)(i)} \{ -y_1^{(i)} + y_1^{(i)}\alpha_1^{(3)(i)} + \alpha_1^{(3)(i)} - y_1^{(i)}\alpha_1^{(3)(i)} \}
\end{aligned} \tag{A.38}$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \alpha_1^{(3)(i)} - y_1^{(i)} = \delta_1^{(3)(i)}$$

Similarly

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \frac{1}{2} \sum_{i=1}^2 \alpha_2^{(2)(i)} \delta_1^{(3)(i)} \rightarrow \frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} = \frac{1}{2} \sum_{i=1}^2 \alpha_j^{(2)(i)} \delta_k^{(3)(i)} \tag{A.39}$$

with

$$\delta_1^{(3)(i)} = \alpha_1^{(3)(i)} - y_1^{(i)}$$

We have now computed six partial derivatives of the cost function $J(\theta)$

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(2)}} = \frac{1}{2} \sum_{i=1}^2 \alpha_j^{(2)(i)} \delta_k^{(3)(i)}, \quad k = 1,2 \text{ and } j = 1,2,3 \quad (\text{A.40})$$

since we know $\alpha_j^{(2)(i)}$ from forward propagation Eqs. (A.29), (A.30) and $\delta_k^{(3)(i)} = \alpha_1^{(3)(i)} - y_1^{(i)}$ with $\alpha_1^{(3)(i)}$ from forward propagation Eq. (A.31), (A.32).

A.2.4 Partial derivatives of previous layers

Similarly with the Section above, the following holds (same proof as above)

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(1)}} = \frac{1}{2} \sum_{i=1}^m \alpha_j^{(1)(i)} \delta_k^{(2)(i)}, \quad k = 1,2,3 \text{ and } j = 1,2 \quad (\text{A.41})$$

We will appose the proof from the derivative above.

Backpropagation starts in the last layer L and successively moves back one layer at a time. For each visited layer it computes the so called error:

$$\frac{\partial \hat{f}(\theta)_i}{\partial z_j^{(2)(i)}}$$

Using the chain rule :

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}} = & \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_1^{(3)(i)}} \frac{\partial \alpha_1^{(3)(i)}}{\partial z_1^{(3)(i)}} \frac{\partial z_1^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{22}^{(1)}} \right) \\ & + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_2^{(3)(i)}} \frac{\partial \alpha_2^{(3)(i)}}{\partial z_2^{(3)(i)}} \frac{\partial z_2^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{22}^{(1)}} \right) \\ & + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_3^{(3)(i)}} \frac{\partial \alpha_3^{(3)(i)}}{\partial z_3^{(3)(i)}} \frac{\partial z_3^{(3)(i)}}{\partial \alpha_2^{(2)(i)}} \frac{\partial \alpha_2^{(2)(i)}}{\partial z_2^{(2)(i)}} \frac{\partial z_2^{(2)(i)}}{\partial \theta_{22}^{(1)}} \right) \end{aligned}$$

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_{31}^{(1)}} &= \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_1^{(3)(i)}} \frac{\partial \alpha_1^{(3)(i)}}{\partial z_1^{(3)(i)}} \frac{\partial z_1^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{31}^{(1)}} \right) \\
&\quad + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_2^{(3)(i)}} \frac{\partial \alpha_2^{(3)(i)}}{\partial z_2^{(3)(i)}} \frac{\partial z_2^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{31}^{(1)}} \right) \\
&\quad + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_3^{(3)(i)}} \frac{\partial \alpha_3^{(3)(i)}}{\partial z_3^{(3)(i)}} \frac{\partial z_3^{(3)(i)}}{\partial \alpha_2^{(2)(i)}} \frac{\partial \alpha_2^{(2)(i)}}{\partial z_2^{(2)(i)}} \frac{\partial z_2^{(2)(i)}}{\partial \theta_{31}^{(1)}} \right) \\
\frac{\partial J(\theta)}{\partial \theta_{32}^{(1)}} &= \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_1^{(3)(i)}} \frac{\partial \alpha_1^{(3)(i)}}{\partial z_1^{(3)(i)}} \frac{\partial z_1^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{32}^{(1)}} \right) \\
&\quad + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_2^{(3)(i)}} \frac{\partial \alpha_2^{(3)(i)}}{\partial z_2^{(3)(i)}} \frac{\partial z_2^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{32}^{(1)}} \right) \\
&\quad + \left(\frac{\partial \hat{f}(\theta)_i}{\partial \alpha_3^{(3)(i)}} \frac{\partial \alpha_3^{(3)(i)}}{\partial z_3^{(3)(i)}} \frac{\partial z_3^{(3)(i)}}{\partial \alpha_2^{(2)(i)}} \frac{\partial \alpha_2^{(2)(i)}}{\partial z_2^{(2)(i)}} \frac{\partial z_2^{(2)(i)}}{\partial \theta_{32}^{(1)}} \right)
\end{aligned}$$

Let's take a closer look at one of the terms, $\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}}$.

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} &= \left(\delta_1^{(3)(i)} \frac{\partial z_1^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right) + \left(\delta_2^{(3)(i)} \frac{\partial z_2^{(3)(i)}}{\partial \alpha_1^{(2)(i)}} \frac{\partial \alpha_1^{(2)(i)}}{\partial z_1^{(2)(i)}} \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right) \\
&\quad + \left(\delta_3^{(3)(i)} \frac{\partial z_3^{(3)(i)}}{\partial \alpha_2^{(2)(i)}} \frac{\partial \alpha_2^{(2)(i)}}{\partial z_2^{(2)(i)}} \frac{\partial z_2^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right)
\end{aligned}$$

We will also calculate the above derivatives for $\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}}$, $\frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}}$, $\frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}}$, $\frac{\partial J(\theta)}{\partial \theta_{31}^{(1)}}$, $\frac{\partial J(\theta)}{\partial \theta_{32}^{(1)}}$

Also we have already mention that the partial derivatives are the activation functions. So it will be:

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} &= \left(\delta_1^{(3)(i)} \theta_{11}^{(2)} \alpha_1^{(2)(i)} (1 - \alpha_1^{(2)(i)}) \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right) \\
&\quad + \left(\delta_2^{(3)(i)} \theta_{21}^{(2)} \alpha_2^{(2)(i)} (1 - \alpha_2^{(2)(i)}) \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right) \\
&\quad + \left(\delta_3^{(3)(i)} \theta_{31}^{(2)} \alpha_3^{(2)(i)} (1 - \alpha_3^{(2)(i)}) \frac{\partial z_2^{(2)(i)}}{\partial \theta_{11}^{(1)}} \right)
\end{aligned}$$

We should also calculate $\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{31}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{32}^{(1)}}$ as the partial derivative above,
like $\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}}$.

Factoring out the $\frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}}$ term, it will be:

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} = \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \left(\delta_1^{(3)(i)} \theta_{11}^{(2)} \alpha_1^{(2)(i)} (1 - \alpha_1^{(2)(i)}) \right) + \left(\delta_2^{(3)(i)} \theta_{21}^{(2)} \alpha_2^{(2)(i)} (1 - \alpha_2^{(2)(i)}) \right) + \left(\delta_3^{(3)(i)} \theta_{31}^{(2)} \alpha_3^{(2)(i)} (1 - \alpha_3^{(2)(i)}) \right)$$

The $\frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}}$ partial derivative will be replaced each time with an input, in our case

x_1, x_2 .

So,

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} = \frac{\partial z_1^{(2)(i)}}{\partial \theta_{11}^{(1)}} \left(\delta_1^{(2)(i)} \right)$$

We should also calculate $\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{31}^{(1)}}, \frac{\partial J(\theta)}{\partial \theta_{32}^{(1)}}$ as the partial derivative above,
like $\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}}$.

We know $\alpha_j^{(1)(i)}$ from the input values, but we do not know $\delta_k^{(2)(i)}$ $k=1,2,3$. Let's compute it.

We know that:

$$\begin{aligned}
\delta_j^{(2)(i)} &= \frac{\partial J(\theta)_i}{\partial z_j^{(2)(i)}} \\
\delta_j^{(2)(i)} &= \frac{\partial \left\{ \begin{aligned} &[-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] - y_2^{(i)} \log(\alpha_2^{(3)(i)}) \\ &-(1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)}) \end{aligned} \right\}}{\partial z_j^{(2)(i)}} \\
&= \frac{\partial \left\{ \begin{aligned} &[-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] - y_2^{(i)} \log(\alpha_2^{(3)(i)}) \\ &-(1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)}) \end{aligned} \right\}}{\partial \alpha_1^{(3)(i)}} \times \\
&\quad \times \frac{d\alpha_1^{(3)(i)}}{dz_1^{(3)(i)}} \frac{\partial z_1^{(3)(i)}}{\partial \alpha_j^{(2)(i)}} \frac{d\alpha_j^{(2)(i)}}{dz_j^{(2)(i)}} \\
&+ \frac{\partial \left\{ \begin{aligned} &[-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] - y_2^{(i)} \log(\alpha_2^{(3)(i)}) - (1 - y_2^{(i)}) \\ &\log(1 - \alpha_2^{(3)(i)}) \end{aligned} \right\}}{\partial \alpha_2^{(3)(i)}} \\
&\times \\
&\quad \times \frac{d\alpha_2^{(3)(i)}}{dz_2^{(3)(i)}} \frac{\partial z_2^{(3)(i)}}{\partial \alpha_j^{(2)(i)}} \frac{d\alpha_j^{(2)(i)}}{dz_j^{(2)(i)}}
\end{aligned} \tag{A.42}$$

since considering a function $F(x, y)$ where $x = x(t), y = y(t)$

$$\frac{dF}{dz} = \frac{\partial F}{\partial x} \frac{dx}{dt} + \frac{\partial F}{\partial y} \frac{dy}{dt}$$

Then,

$$\begin{aligned}
&\frac{\partial \left\{ \begin{aligned} &[-y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)})] - y_2^{(i)} \log(\alpha_2^{(3)(i)}) \\ &-(1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)}) \end{aligned} \right\}}{\partial \alpha_1^{(3)(i)}} \\
&= \left[-\frac{y_1^{(i)}}{\alpha_1^{(3)(i)}} + \frac{(1 - y_1^{(i)})}{1 - \alpha_1^{(3)(i)}} \right]
\end{aligned} \tag{A.43}$$

Similarly

$$\frac{\partial \left\{ -y_1^{(i)} \log(\alpha_1^{(3)(i)}) - (1 - y_1^{(i)}) \log(1 - \alpha_1^{(3)(i)}) - y_2^{(i)} \log(\alpha_2^{(3)(i)}) - (1 - y_2^{(i)}) \log(1 - \alpha_2^{(3)(i)}) \right\}}{\partial \alpha_2^{(3)(i)}} = \left[-\frac{y_2^{(i)}}{\alpha_2^{(3)(i)}} + \frac{(1 - y_2^{(i)})}{1 - \alpha_2^{(3)(i)}} \right]$$

Furthermore from Eq. (A.35)

$$\frac{d\alpha_1^{(3)(i)}}{dz_1^{(3)(i)}} = \alpha_1^{(3)(i)} (1 - \alpha_1^{(3)(i)}) \quad (\text{A.44})$$

and

$$\frac{d\alpha_2^{(3)(i)}}{dz_2^{(3)(i)}} = \alpha_2^{(3)(i)} (1 - \alpha_2^{(3)(i)})$$

Finally, from Eq. (A.31)

$$\frac{\partial z_1^{(3)(i)}}{\partial \alpha_j^{(2)(i)}} = \theta_{1j}^{(2)} \quad (\text{A.45})$$

$$\frac{\partial z_2^{(3)(i)}}{\partial \alpha_j^{(2)(i)}} = \theta_{2j}^{(2)}$$

and,

$$\frac{d\alpha_j^{(2)(i)}}{dz_j^{(2)(i)}} = \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) \quad (\text{A.46})$$

Substituting Eqs. (A.42) – (A.45) into Eq. (A.41) we obtain

$$\begin{aligned} \delta_j^{(2)(i)} &= \\ &= \frac{\partial J(\theta)_i}{\partial z_j^{(2)(i)}} = \left[-\frac{y_1^{(i)}}{\alpha_1^{(3)(i)}} + \frac{(1 - y_1^{(i)})}{1 - \alpha_1^{(3)(i)}} \right] \alpha_1^{(3)(i)} (1 - \alpha_1^{(3)(i)}) \theta_{1j}^{(2)} \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) \\ &\quad + \left[-\frac{y_2^{(i)}}{\alpha_2^{(3)(i)}} + \frac{(1 - y_2^{(i)})}{1 - \alpha_2^{(3)(i)}} \right] \alpha_2^{(3)(i)} (1 - \alpha_2^{(3)(i)}) \theta_{2j}^{(2)} \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) \end{aligned} \quad (A.47)$$

Performing the multiplications (similarly to Eq. (A-9)) we obtain

$$\begin{aligned} \delta_j^{(2)(i)} &= \delta_1^{(3)(i)} \theta_{1j}^{(2)} \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) + \delta_2^{(3)(i)} \theta_{2j}^{(2)} \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) \\ &\Rightarrow \delta_j^{(2)(i)} = [\delta_1^{(3)(i)} \quad \delta_2^{(3)(i)}] \begin{bmatrix} \theta_{1j}^{(2)} \\ \theta_{2j}^{(2)} \end{bmatrix} \alpha_j^{(2)(i)} (1 - \alpha_j^{(2)(i)}) \\ &\Rightarrow [\delta_1^{(2)(i)} \quad \delta_2^{(2)(i)} \quad \delta_3^{(2)(i)}] \\ &= [\delta_1^{(3)(i)} \quad \delta_2^{(3)(i)}] \begin{bmatrix} \theta_{11}^{(2)} & \theta_{12}^{(2)} & \theta_{13}^{(2)} \\ \theta_{21}^{(2)} & \theta_{22}^{(2)} & \theta_{23}^{(2)} \end{bmatrix} .* [\alpha^{(2)(i)} .* (1 - \alpha^{(2)(i)})] \\ &\Rightarrow \delta^{(2)(i)} = [\delta^{(3)(i)} \theta^{(2)}] .* [\alpha^{(2)(i)} .* (1 - \alpha^{(2)(i)})] \end{aligned} \quad (A.48)$$

Taking the average of the two training samples $i = 1, 2$

$$\delta^{(2)} = \delta^{(3)} \theta^{(2)} .* (\alpha^{(2)} .* (1 - \alpha^{(2)}))$$

Now we can compute the other six partial derivatives of the gradient vector

$$\frac{\partial J(\theta)}{\partial \theta_{kj}^{(1)}} = \frac{1}{2} \sum_{i=1}^m \alpha_j^{(1)(i)} \delta_k^{(2)(i)}, \quad k = 1, 2, 3 \text{ and } j = 1, 2 \quad (\text{A.49})$$

with

$$\delta_k^{(2)(i)} = [\delta_1^{(3)(i)} \quad \delta_2^{(3)(i)}] \begin{bmatrix} \theta_{1k}^{(2)} \\ \theta_{2k}^{(2)} \end{bmatrix} \alpha_k^{(2)(i)} (1 - \alpha_k^{(2)(i)}) \quad (\text{A.50})$$

all known from forward propagation

Thus, for any iteration we can compute the gradient $\nabla J(\theta)$ from the values of θ of the previous iteration and the results of the forward propagation.

$$\nabla J(\theta) = \left[\frac{\partial J}{\partial \theta_{11}^{(1)}}, \dots, \frac{\partial J}{\partial \theta_{32}^{(1)}} \mid \frac{\partial J}{\partial \theta_{11}^{(2)}}, \dots, \frac{\partial J}{\partial \theta_{23}^{(2)}} \right]$$

Using Eqs. (A.39), (A.49), (A.50)

The new values of θ are $[\theta_{11}^{(1)}, \dots, \theta_{32}^{(1)} \mid \theta_{11}^{(2)}, \dots, \theta_{23}^{(2)}]_{\text{new}}$
 $= [\theta_{11}^{(1)}, \dots, \theta_{32}^{(1)} \mid \theta_{11}^{(2)}, \dots, \theta_{23}^{(2)}]_{\text{old}} - a \nabla J$

Appendix B. Implementing Convolution Neural Networks in TensorFlow

In this Appendix contains techniques about image classification through Convolution Neural Network. Firstly, a model for binary classification with less convolutional layers is presented. Secondly, a model for multiple classifications is discussed which contain a lot of convolutional networks. Finally, a model for data augmentation shows techniques for image processing.

B.1 A model for binary classification

Below in Fig. A.1 we illustrate the implementation of a Neural Network model for binary classification in code. The NN has 3 layers, one layer is the input the second layer is the hidden one and the third layer is the output. The hidden layer has 1,024 hidden units and we use Relu activation as we have mentioned in the text above. The output layer has 1 unit. The loss function is binary crossentropy and the optimizer is RMSprop.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.optimizers import RMSprop

# Flatten the output layer to 1 dimension
x = layers.Flatten()(last_output)
# Add a fully connected layer with 1,024 hidden units and ReLU activation
x = layers.Dense(1024, activation='relu')(x)
# Add a final sigmoid layer for classification
x = layers.Dense (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)

model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['accuracy'])
```

```
history = model.fit(  
    train_generator,  
    validation_data = validation_generator,  
    epochs = 20)
```

Figure B. 1 CNN with one Dense

B.2 A model for multiple classification

The coded model below is utilized for 3 class classification. The model has 4 convolutional and maxpooling layers. 64 filters are utilized, of size 3×3 . Then, filters are increased to 128. The maxpooling layer size is 2×2 . The activation function for convolutional layers is Relu. The Neural Network has 3 layers. First layer is the input, second layer is the hidden one and third layer is the output. The input layer has images with 150×150 size and 3 byte color (Red, Green, Blue). The input images are first Flattened and then the Dropout technique is used (see text in Section 2.2) for definitions). The hidden layer has 512 units and Relu activation is used. The output layer has 3 units, since there are 3 classes. The output activation function is SoftMax, which is more suitable for multiclass classification than Relu. The loss function is categorical crossentropy and the optimizer is RMSprop.

```
import tensorflow as tf  
import keras_preprocessing  
  
model = tf.keras.models.Sequential([  
    # Note the input shape is the desired size of the image 150x150 with 3 bytes color  
    # This is the first convolution  
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(150, 150, 3)),  
    tf.keras.layers.MaxPooling2D(2, 2),  
    # The second convolution  
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),  
    tf.keras.layers.MaxPooling2D(2,2),
```

```
# The third convolution
tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The fourth convolution
tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# Flatten the results to feed into a DNN
tf.keras.layers.Flatten(),
tf.keras.layers.Dropout(0.5),
# 512 neuron hidden units
tf.keras.layers.Dense(512, activation='relu'),
tf.keras.layers.Dense(3, activation='softmax')
])

model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

Figure B. 2 A model used for multiple classification.

B.3 A model for data augmentation

The code below is used to apply the data augmentation technique. Firstly, we prepare the training and validation data before they are used in the neural network process. Some of the training images are used to rescale, rotate 40%, width shift range 20%, height shift range 20%, shear range 20%, zoom 20%, horizontal flip and fill mode. After data augmentation, the training images have 150×150 size, the class mode is categorical, since there are 3 classes, and the batch size of images is 126. The validation data has 3×3 size, the class mode and the batch size are the same as the training data.

```
import tensorflow as tf
import keras_preprocessing
from keras_preprocessing import image
```

```
from keras_preprocessing.image import ImageDataGenerator

TRAINING_DIR = "/tmp/rps/"
training_datagen = ImageDataGenerator(
    rescale = 1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

VALIDATION_DIR = "/tmp/rps-test-set/"
validation_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(150,150),
    class_mode='categorical',
    batch_size=126
)

validation_generator = validation_datagen.flow_from_directory(
    VALIDATION_DIR,
    target_size=(150,150),
    class_mode='categorical',
    batch_size=126
)
```

Figure B. 3 Data augmentation technique.

Appendix C. Annotating images with ground truth bounding boxes

Image annotation is defined as the task of annotating an image with labels for supervised machine learning. Labels are chosen to provide the network with information about what is shown in the image. In this thesis the annotated traffic lights classes, such as green, red and yellow, are used for training and validation.

The majority of computer vision models are created using image annotation tools. The latter usually involve manual work from users, sometimes with computer-assisted help. Users define the labels, known as “classes”, and provide the image-specific information to the computer vision model. After the model is trained, it will predict and detect those features in new images that have not been annotated yet (Boesch, 2021).

There are many free tools for image annotation tasks. Some of them are (Morgunov, 2021):

- VGG Image Annotator (VIA)
- CVAT – Computer Vision Annotation Tool
- Labellmg
- Visual Object Tagging Tool (VoTT)

Labellmg¹³ is a graphical image annotation tool which is open source. This tool is selected for creating label ground truth bounding boxes in the image dataset, especially in cases of datasets over 10,000 images.

To install the tool, one needs to go to the Labellmg 1.8.5¹⁴ website, which describes in detail the installation steps. After installation, the Labellmg interface (see Fig B.1) is displayed on the screen. For uploading the images from the dataset, the photos are selected by selecting the “Open Dir” command. Next, the user selects the correct folder (see Fig B.2).

¹³ Github repository of darrenl tzutalin: <https://github.com/tzutalin/labellmg>

¹⁴ <https://pypi.org/project/labellmg/>

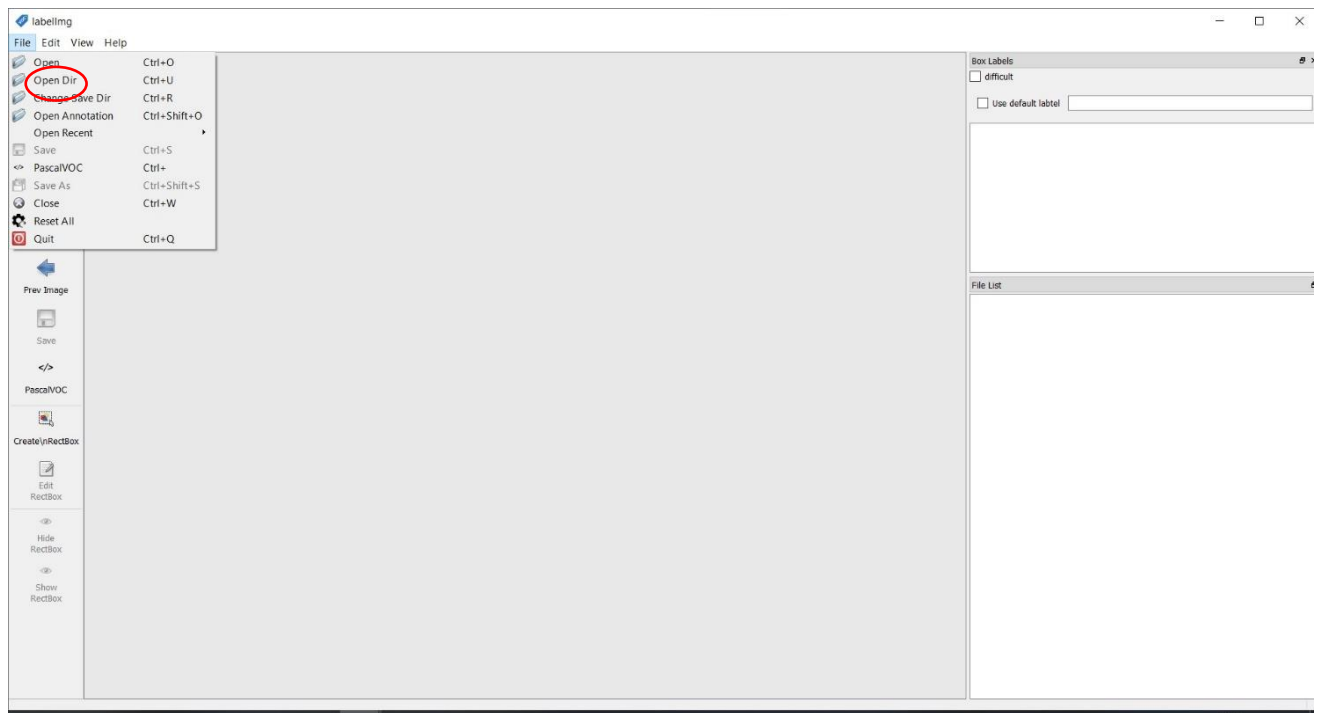


Figure C. 1 Open Directory

Subsequently, the location to save the annotation file is selected. This is done by the command Change Save Dir.

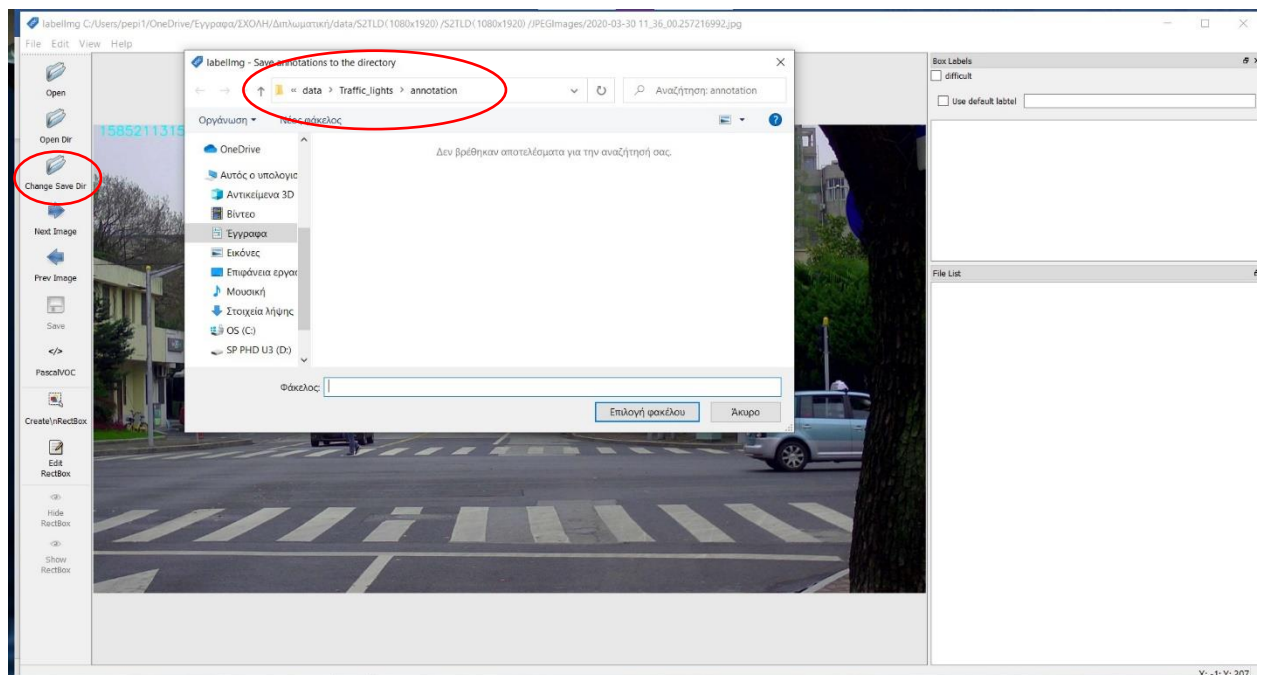


Figure C. 2 The location of files

After that, the annotation images will be saved in a format recognized by the YOLO algorithm. The save format should be changed from the Pascal VOC to YOLO.

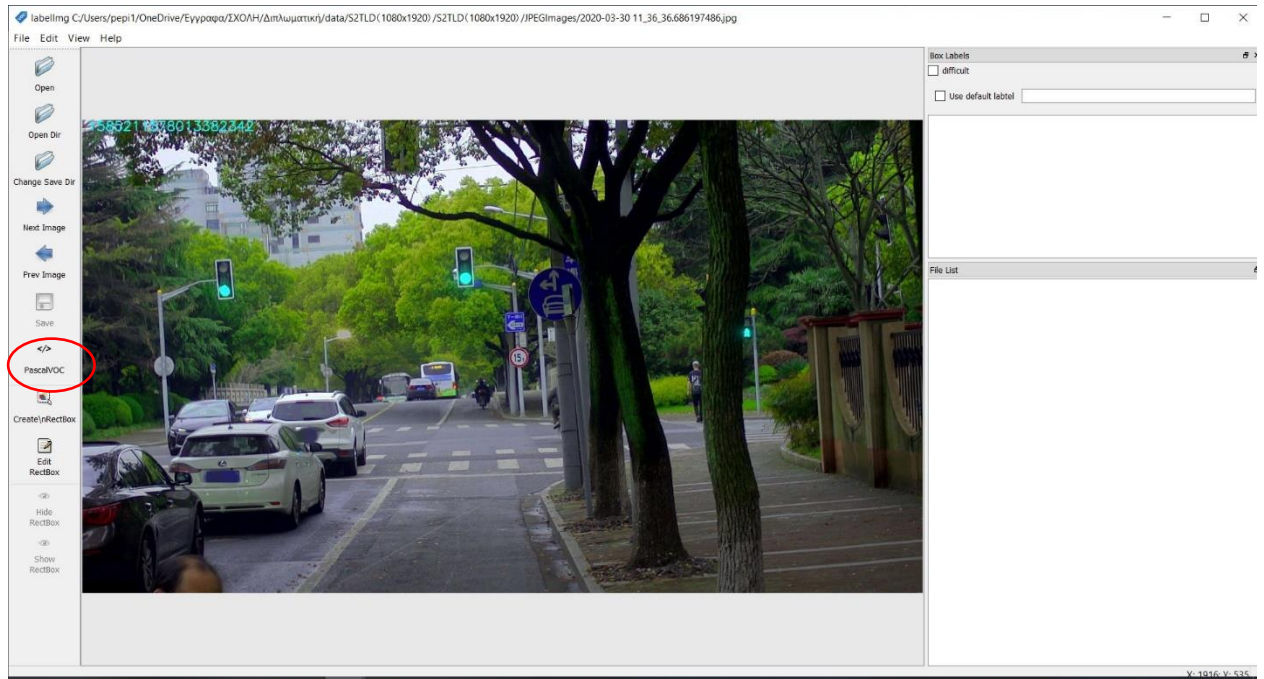


Figure C. 3 The save format (Pascal/VOC)

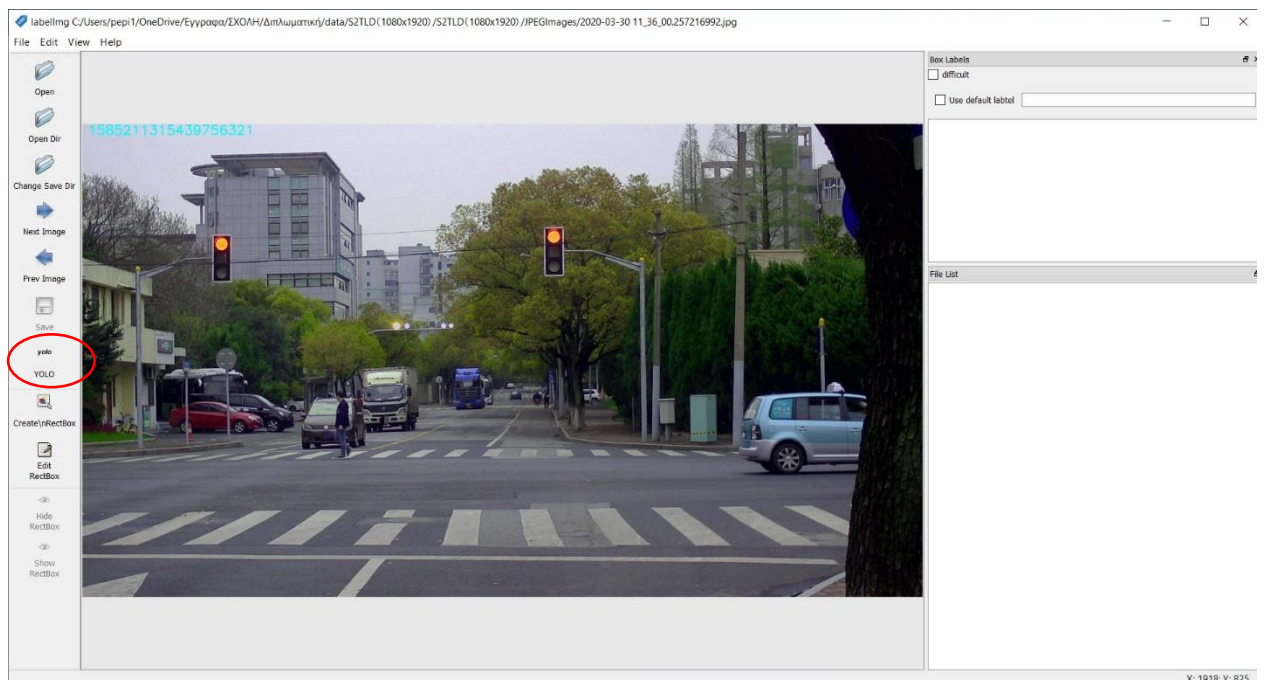


Figure C. 4 The save format (YOLO)

In the final step, the label ground truth bounding boxes are created on the custom images. The ground truth bounding boxes are drawn by the Create\nRectBox button. Now, the ground truth bounding boxes can be drawn over the image.

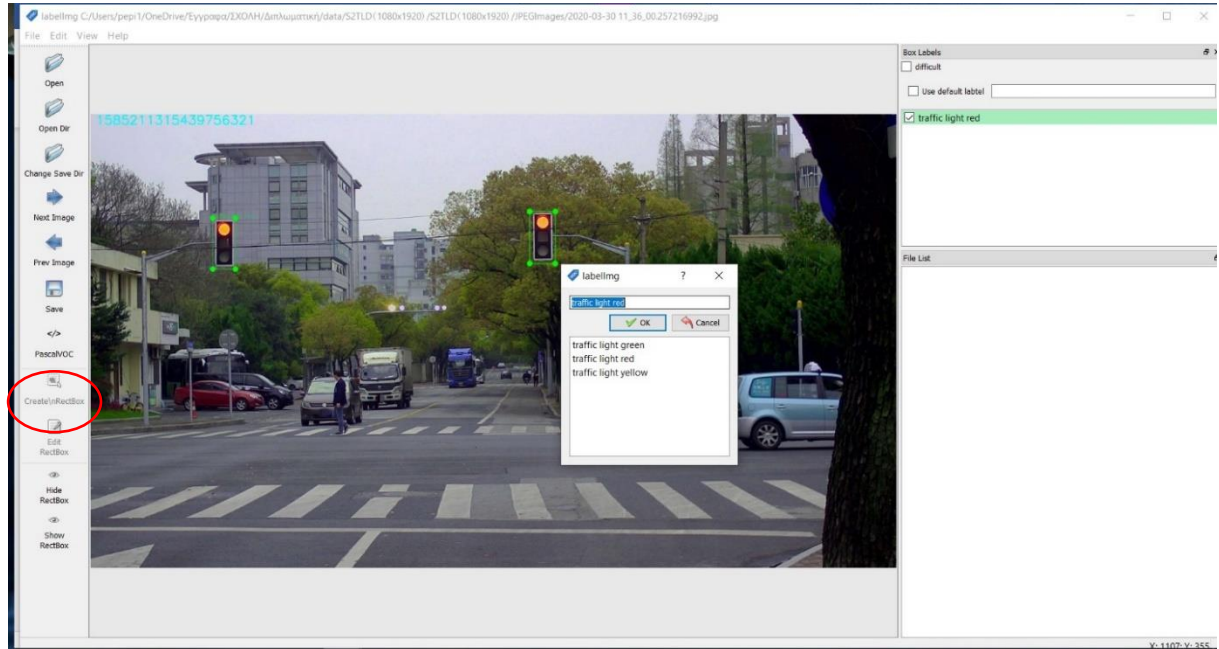


Figure C. 5 Create\nRectBox button for drawing the ground truth bounding box

Each time drawing the box is completed, a new Labelling window pop up shows up. The object name in the text field of Fig. B.5 is defined by the user. Once labeling objects in the image is completed, the Save button on the left menu or command ctrl+s (of keyboard) should be clicked to save the annotation images in the folder.

The final custom dataset is ready for object detection. The annotation tool extracts the information that it needs (txt files¹⁵) that contain the coordinates of ground truth bounding boxes (scaled from 0 to 1) and the classes of the objects, which are included in the image. In our case, these objects are traffic lights.

¹⁵ The txt file should be saved in the same directory, and the same name as the image.