MICROSCOPIC IMAGE CELL COUNTING USING CONVOLUTIONAL NEURAL NETWORKS

A THESIS SUBMITTED TO

THE FACULTY OF ARCHITECTURE AND ENGINEERING

OF

EPOKA UNIVERSITY

BY

ALEKS TARE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2020

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled **“Microscopic Image Cell Counting using Convolutional Neural Networks”** and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

____

Dr. Ali Osman Topal (Head of Department)
Date: July 13, 2020

Examining Committee Members:

Dr. Ali Osman Topal	(Computer Engineering)	________________
Assoc. Prof. Dr. Dimitrios A. Karras	(Computer Engineering)	________________
Dr. Arban Uka	(Computer Engineering)	________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name Surname: Aleks Tare

Signature: ______________

ABSTRACT

MICROSCOPIC IMAGE CELL COUNTING USING CONVOLUTIONAL NEURAL NETWORKS

Tare, Aleks

M.Sc., Department of Computer Engineering

Supervisor: Dr. Arban Uka

As the field of automation is moving forward at ever-faster rates, cell counting and classification is an omnipresent yet repetitive task that would benefit greatly from this field. The counting of contiguous cells in a specific area could provide crucial contribution to work done in clinical trials. Cell counting, sadly, is most often conducted manually by humans and can be time and resource consuming.

Due to cells touching each other, a non-uniform background, shape and size variations of cells, and different techniques of image acquisition, the task becomes even more difficult. In this paper we describe a convolutional neural network approach, using a Faster-RCNN architecture later also combined with a U-Net neural network, for cell counting and possibly segmentation in a raw microscopic picture.

Key words: *machine learning, microscopy, faster-rcnn, classification, cell counting*

ABSTRAKT

NUMERIMI I QELIZAVE NE IMAZHET MIKROSKOPIKE DUKE PERDOR RRJETAT NEURALE KONVOLUCIONARE

Tare, Aleks

Master Shkencor, Departamenti i Inxhinierise Kompjuterike

Udhëheqësi: Dr. Arban Uka

Ndërkohë që fusha e automatizmit po ecën përpara me ritme gjithnjë e më të shpejta, numërimi dhe klasifikimi i qelizave është një detyrë që do të përfitonte shumë nga kjo fushë pasi ky i fundit paraqet sfida te mundimshme për stafin mjekësor. Numërimi i qelizave të ngjitura me njëra-tjetrën, në një zonë specifike, mund të sigurojë një kontribut thelbësor në punën e kryer në provat klinike. Numërimi i qelizave, për fat të keq, shpesh herë kryhet në mënyrë manuale nga njerëzit dhe si veprim konsumon tepër kohë dhe burime.

Për shkak të qelizave që prekin njëra-tjetrën, sfondeve jo-të-njëtrajtshëm, ndryshime të formës dhe madhësisë së qelizave, si dhe teknikave të ndryshme të fotografimit të imazhit, kjo detyrë bëhet edhe më e vështirë. Në këtë punim ne përshkruajmë një qasje të rrjetit neural konvolucionar, duke përdorur një arkitekturë 'Faster-RCNN' të kombinuar më vonë edhe me një rrjet neural 'U-Net', për numërimin e qelizave dhe mundësisht segmentimin e tyre në një imazh të papërpunuar mikroskopik.

Fjalët kyçe: *machine learning, mikroskopi, faster-rcnn, klasifikim, numerim qelizash*

ACKNOWLEDGEMENTS

I would first like to express my gratitude to my thesis advisor Dr. Arban Uka of the Computer Engineering Department at Epoka University. Whenever I ran into a trouble spot or had a question about my research Prof. Uka was always willing to help. His valuable advice and suggestions helped steer me into the best learning path during my studies in Epoka. I would also like to thank Alba Tujani who was involved in the labelling of images for this research project. Without her relentless participation and input, the labelling of over five-thousand cells could not have been conducted.

TABLE OF CONTENTS

ABSTRACT....iv

ABSTRAKT v

ACKNOWLEDGEMENTS....vi

TABLE OF CONTENTS.... vii

LIST OF FIGURES ix

LIST OF TABLES....x

CHAPTER 1 1

1. INTRODUCTION....1

   1.1 Cell Counting 1

   1.2 Thesis Objective & Scope of Works....1

   1.3 Organization of the Thesis 2

CHAPTER 2 3

2. LITERATURE REVIEW....3

   2.1 Medical Image Analysis....3

   2.2 Challenges of this research....4

   2.3 Microscopy Image Analysis....6

   2.4 Use of Neural Networks in Medicine Pathology 6

   2.5 Segmentation Techniques in Medical Imaging 7

CHAPTER 3 8

3. FROM MACHINE LEARNING TO NEURAL NETWORKS 8

   3.1 Batch Learning 9

   3.2 Artificial Neurons....9

   3.3 Logical Computations with Neurons 10

   3.4 Computer Vision Techniques....12

CHAPTER 4 14

4. METHODOLOGY 14

4.1 VGG – a standard CNN architecture....14
4.2 Faster R-CNN – a powerful object detection model....16
4.2.1 Region Proposals Network (RPN)....16
4.2.2 Mean Average Precision Score....17
4.2.3 Classification inside Faster-RCNN....18
4.2.4 RoI Pooling....19
4.2.5 Intersection over Union Threshold (IoU)....20
4.3 Labelling the Dataset....21
CHAPTER 5....22
5. RESULTS AND DISCUSSIONS....22
5.1 Initial Model 03....22
5.2 Improvements in Model 04....25
5.3 Histogram Matching Preprocessing in Model 05....27
5.4 Selective-Histogram Matching Classification in Model 06....31
CHAPTER 6....34
6. CONCLUSIONS....34
6.1 Conclusions....34
6.2 Future Work....34
7. REFERENCES....35
8. APPENDIX....39

LIST OF FIGURES

Figure 1. Crowded Cell Image....4
Figure 2. Out of Focus Cells in an Image....5
Figure 3. Different Contrast Extremes in the Dataset....5
Figure 4. Simple Neurons Illustrated as Logic Gates....10
Figure 5. Anatomy of the Perceptron....11
Figure 6. Activation Functions & their Derivatives....12
Figure 7. Simple CNN Architecture....13
Figure 9. VGG-16 Architecture....15
Figure 10. CNN Feature Map Flow....17
Figure 8. RoI Max Pooling Example....19
Figure 11. LabelImg User Interface....21
Figure 12. Model 03 Test Results....23
Figure 13. M03 Total Loss & Other Graphs (above)....24
Figure 14. Model 04 Test Results....25
Figure 15. M04 Total Loss & other graphs (above)....26
Figure 16. Difference of contrast inside a sample....27
Figure 17. Histogram Matching Technique....28
Figure 18. Model 05 Test Results....28
Figure 19. M05 Total Loss & other graphs (above)....29
Figure 20. Images before Selective Histogram Matching....31
Figure 21. Images after Selective Histogram Matching....31
Figure 23. M06 Total Loss & other graphs (above)....32
Figure 22. M06 Test Results....33

LIST OF TABLES

Table 1- Model 03 Results 22
Table 2- Model 04 Results 25
Table 3- Model 05 Results 30
Table 4- Model 06 Results 33

CHAPTER 1

1. INTRODUCTION

1.1 Cell Counting

Medical images provide health professionals in-depth knowledge of the inner operation of a person's body, as well as the functions of their organs and tissues. This knowledge is often key to the clinical analysis of a person's situation or the implementation of life saving medical interventions. Medical imaging technology including acquisition and analysis is constantly evolving and we need knowledgeable and highly skilled practitioners to ensure the best care for the patient. In this work, we will analyze state-of-the-art methods for detection such as Faster-RCNN [1] and image segmentation techniques such as U-Net [2] for each respective stage of our research to detect cells and determine the area covered by them. This evolution comes at a cost and that is the effort needed by the medical staff to analyze and generate results out of these images. One of the issues treated in this thesis is the counting of the cells in a microscope image. This task is usually done manually and copious amounts of time are required for the medical staff to finish it.

1.2 Thesis Objective & Scope of Works

In this paper, we hope to achieve an automated cell counting technique by utilizing pre-existing convolutional neural networks. The research group aims to create a successful deep learning model, which accurately detects the location of each cell present in the images provided in our dataset. Firstly, a literature review is conducted in order to determine which model suits best the task at hand, and also later on to analyze the possibilities of linking our model with a segmentation task in order to accurately detect the area covered by each cell. In addition, the recent methods of pre-processing and regularization will be analyzed during this process. Finally, the scope of the thesis covers multiple experiments with different preprocessing techniques such as

- template matching and contrast correction

- alternating hyperparameters for the model

The goal of this research is to achieve the best performance possible with a CNN model for the detection of the maximum number of cells we can in a microscope image.

1.3 Organization of the Thesis

This thesis is organized in six chapters with each one representing a part of the work conducted from the beginning of the research up until the results and conclusions.

- Chapter 1 presents the introduction to the motivations behind this work and its purposes.
- Chapter 2 explores the correlated researches in the field of medical image analysis through different approaches such as classification and object detection by utilizing neural networks.
- Chapter 3 holds all the necessary architecture information that supports all the work in this thesis. It covers some state-of-the-art techniques in object detection models and their evaluation methods.
- Chapter 4 describes the methodology used for the experiments conducted and the resources applied during these experiments.
- Chapter 5 includes all the information about the parameters of each experiment and the interpretation of their results.
- Chapter 6 interprets the results, discusses the conclusions and recommends future work for research on this topic.

This thesis also includes sections for easier navigation such as the List of Tables, List of Figures and Table of Contents.

CHAPTER 2

2. LITERATURE REVIEW

2.1 Medical Image Analysis

The rapidly developing optical microscopy area has evolved over the past few years from depending on conventional photomicrography using emulsion-based film to one where state-of-the-art digital images are produced as a result. These images can be transformed and altered through various processes such as changing their spatial or gray-level resolution, contrast manipulation and stretching, gamma correction, noise removal, background subtraction and so on.

Implanting of biocompatible materials has been used for many decades. Tooth implants, silicone implants, hip replacements have been used for a long time and the use of them is expected to rise. The market share is predicted to be 130 billion dollars pointing to the importance of the research relating the use of biomaterials. The most important aspect in the use of biomaterials is the prediction of unwanted complications that may arise at a personalized level. A certain biomaterial may have shown to be compatible over the course of a certain testing phase, but complications in new patients cannot be completely ruled out. Because of this, a personalized level analysis of the biocompatibility of materials is essential such as Dollinger *et al* [3].
Lab scientists to stimulate and mimic tissue growth are studying biomaterials. These so-called “regenerative functional porous materials” are mixed with living cell cultures so they can stimulate mechanisms to work for the regeneration of a specific body tissue. However, the tests and clinical trials must be conducted in laboratory environments such as incubators before they are tested *in vivo*. This requires thorough observation of these cultures, by daily or weekly cycles, in order to detect anomalies such as cells dying or turning cytotoxic in reaction to the biomaterial. [4], [5] Analyzing these images manually is not only expensive and lengthy but it also suffers from the variation of observers-interpretation.

A little earlier, computer vision researches thought that employing a computer to tell the difference between a cat and a dog would be almost impossible, whereas now this can be achieved at an accuracy of higher than 99%. This is image classification, which includes the labeling of images, in two or more classes and then identifying an image after a training of a certain neural network. In the recent years, researchers have also achieved object detection using artificial intelligence that is to look at an image and try to find all the objects in that image, assigning bounding boxes around these objects and label exactly what those objects are. Similar researches conduct the same task but with different types of cells. [6] [7] [8]

In order to alleviate human effort in the sector of biomaterial testing, we can apply image detection and classification techniques to automatize the process of detecting these cells, counting them and if possible classify them into healthy and unhealthy categories.

2.2 Challenges of this research

One of the initial challenges faced in this research was the labelling of the images. A raw dataset of images was provided and the initial labelling was done as manual work over the course of 3 months in a total of approx. 100 hours to label 86 images.

Another great challenge posed by this research can be seen in the image below. The cells can be very crowded in an image and are usually touching other cells. With a high number of cells touching each other, it is hard even for the human eye to distinguish cells from the background.

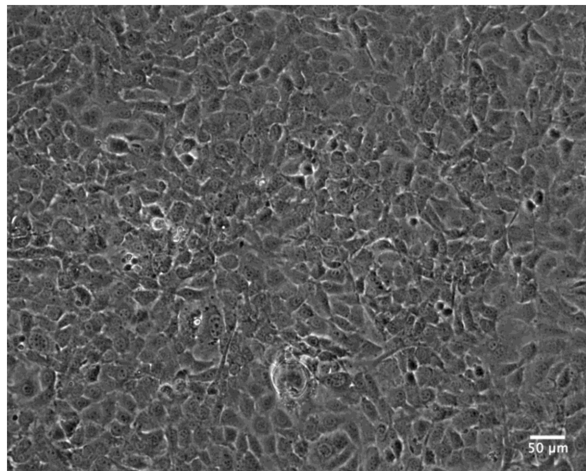


Figure 1. Crowded Cell Image

Another challenge is the difference in contrast and lighting of each image. Compared to the image above, the following picture demonstrates a huge change in focus of the image, which in terms can throw off the neural network when detecting cells. (False Positive/False Negatives cells are detected)

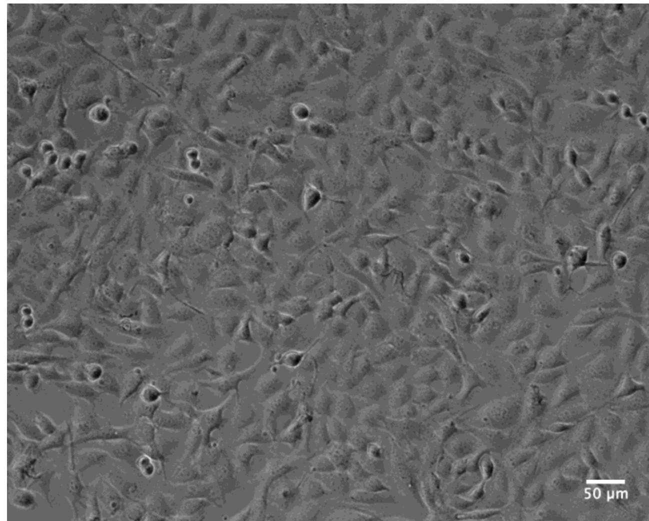


Figure 2. Out of Focus Cells in an Image

One of the major challenges belonging to the group of image qualities (and which is effectively treated in this paper) is that of the contrast differences between images. As we can see from the example below, two extremes of high and low, contrasts are present throughout the entire dataset. This would require a thorough preprocessing technique in order to create a better model for the cell counting task such as [9]

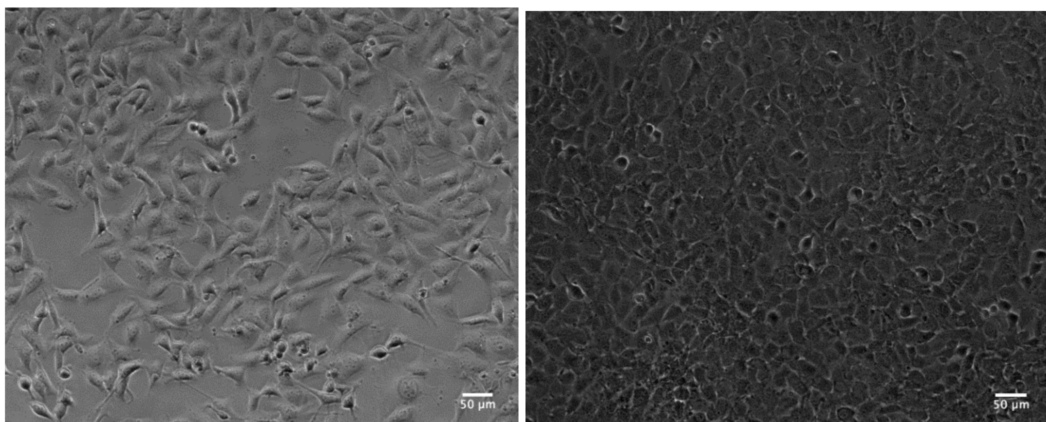


Figure 3. Different Contrast Extremes in the Dataset

2.3 Microscopy Image Analysis

One of the main problems in microscopic anatomy image analysis is to count the cells in the image. This consists in not only the complexity and variety of algorithm being applied, but also it is a process, which is held back by the characteristics of microscopic images such as cells touching each other, background disorders, shape and size variations of cells, and different techniques of image acquisition. We must also note that these histological images come in high resolutions, thus consuming many computational resources.

2.4 Use of Neural Networks in Medicine Pathology

Convolutional Neural Networks (CNNs) have been widely used for image processing purposes not only in medical imaging, but also in fields of security, space exploration images, autonomous driving, fake news detection and many more. [10] [11], [12] It is implemented to also adapt to real time images which can change dynamically, such as sign language detection and translation. [13] However, we notice that the most impactful field is that of improving medical diagnosis. Innovative designs of CNNs have shown great progress in the field of dealing with melanoma cancer cells by using a committee of CNNs [14] or the detection of colorectal cancer in real time colonoscopy images. [15] [16] [8] These have shown a superior efficiency level for Fast-RCNN implementations compared to similar classifiers in this field. Other researchers still applied their version of Deep Convolutional Neural Networks in order to diversify the approach for a specific type of detection in medical imaging. Detect Net from the Caffe package was used for detection of astrocytes involved in different brain pathologies. [17]–[19] Besides the complex implementations of FRCNN in Caffe or Tensorflow, it is worth noting the simplified join-training scheme of the pipeline, which unifies functionalities of both these libraries. [20] We may also mention that this is not the first time U-Net has joined forces with Fast-RCNN since the use of this latter in a 3D Faster R-CNN model, adapted in the architecture of U-Net for the detection of pulmonary nodules to prevent lung cancer. [21] Similar methods include generation of a mask to identify cells in an image. Of particular note is the Mask-RCNN algorithm designed by Facebook AI Research (FAIR). [22] Multiple augmentation methods have been reviewed such as [23]

2.5 Segmentation Techniques in Medical Imaging

The next challenging step is segmentation of these detected regions, and the most-valued method in the field is the state-of-the-art approach of U-Net. [2] This model finds usages in various medical fields, especially in MRI Imaging. By using multiple 2D U-Nets for analyzing MR brain images in order to diagnose glioma, a type of malignant brain tumor [18], [24], or an "autoencoder-regularized 3D-CNN" with three stages of encoding the low dimensional input, reconstructing it and finally segmentation through U-Net for the same problem of brain tumors, providing this way a memory efficient approach. [19] U-Net has also been evaluated in 2D and 3D architectures against DenseNet for Cardiac MRI Image Segmentation, to distinguish between left ventricle and right ventricle, with the 2D architecture having the upper accuracy of generalization. [25] DenseU-Net is based on encoding, connecting and decoding dense blocks and it provides substantial gains over the baseline U-Net model in terms of Dice Score improvement. [17] A similar approach is that of AD-UNet for vessel segmentation purposes in retinal images, except that it also includes the attention mechanism, of which the encoder and decoder components are added with dense blocks. [12] Also, in the same field another approach introduces a method in which the pooling layers of the encoder part are replaced by strided convolutional layers. [1] Other researchers have preferred a raw implementation of CNN to achieve the same task on eye vasculature. [26] [27] [28] On the other hand, 3D U-Net performs very well for segmentation of thoracic Organs-At-Risk by using cropped 3D images. It has shown potential for eventual clinical adoption of deep learning in radiation treatment planning due to improved accuracy and reduced cost for OAR segmentation. [29] For knee cartilage segmentation the original UNet with 40 channels and a version of UNet with dilated convolutions are ensembled in (Duarte, 2019) [30] The dilation model helps achieve a larger field of view while the original model improves the accuracy of the smaller regions. In addition, other methods such as Random Forest Semantic Classification prove to be better in the field of brain MRI for the problem of weak signals produced by bone and air tissues. [18] The illumination and contrast of cell images varies greatly. To reduce this variance and enhance the contrast, a method of normalizing each image by first subtracting the minimum intensity value of the image is proposed by Gao et al [16].

CHAPTER 3

3. FROM MACHINE LEARNING TO NEURAL NETWORKS

In the paper published by Geoffrey Hinton [31] in 2006, it is demonstrated the training of a deep neural network (DNN) able to recognize handwritten digits even with the highest precision possible at that time (>98%). This technique was later known as "Deep Learning." A DNN is a cut down version of our brain's cerebral cortex, which in this case is made of a stack of layers of artificial neurons. In the late 1990s, it was considered a very hard and tedious task to train a DNN and most researchers abandoned the idea. This paper revitalized the scientific community's interest and it would not be long that other papers would join to show that Deep Learning could surpass Machine Learning techniques even in most complicated tasks. However, a great help were the advancements made in computing power and speed and the larger ever-growing amount of data we feed into the Internet today. This newborn interest soon stretched into many other capacities of Machine Learning.

A decade or so later, Machine Learning is at the very top of the productive processes happening inside every computer: it is the core of what people call magic of the computers nowadays, classification of web results, speech recognition conducted live in your handheld device, movie recommendation and defeating world class champions in games such as Go. It is already working and improving on driving our cars.
It is possible to classify different types of Machine Learning in extensive categories, since so many of them exist. This is done in accordance with the following criteria: If human supervision has been present during the training (supervised, unsupervised, semi supervised, and Reinforcement Learning). Another way of categorizing them is based on the ability to learn incrementally on the fly (online versus batch learning). We can combine these criteria in any way we like, since they are not exclusive. As an example, we can analyze a state-of-the-art spam filter. This filter can learn on the fly by using a deep neural network model. The model itself utilizes examples of spam and ham to train itself. This learning system is known to be an online, model-based, supervised one. (Benjamin Planche, 2019)
We can also classify Machine Learning systems by analyzing the type and quantity of supervision the models can have during the process of training. Unsupervised learning,

supervised learning, semi supervised learning, and reinforcement Learning are some major categories. In supervised learning, you feed a training set that includes the desired solutions to the algorithm. These solutions are known as labels. Classification is a very typical supervised learning task. As we explained above in the spam filter, we use example emails and we classify them as spam or ham. Then the model trains itself and can learn how to classify other emails. Predicting numerical values, as targets, for example the price of a car, is another typical task. In this case, we provide what we call predictors. Predictors are a group of different features, such as age, mileage, color, brand, etc. This type of function can also be known as regression. (Benjamin Planche, 2019)

3.1 Batch Learning

In this type of learning, the system is not able to learn incrementally: we must use all the available data to train it. It is usually not done online as it generally involves a huge quantity of time and resources to compute. The model is firstly trained, and then is able to run without learning anymore, while being launched into production and applying what has already been learned. We call this offline learning. Let us say that we want to add new data to a currently live batch learning system. In this case, a new version of the system needs to be trained from the beginning using the complete dataset. The full dataset contains both old and new data. Then we need to replace the old model with the new one. The solution may be simple, but there is a downside to using the full dataset to train. This can require prolonged hours and a new system would generally be trained every 24 hours or even weekly. In addition, a huge amount of computing resources such as CPU, disk space, memory space, inputs, outputs and more are required. Therefore, we can lean on using algorithms capable of learning incrementally as a better option. [32]

3.2 Artificial Neurons

McCulloch and Pitts firstly introduced the Artificial Neural Networks in 1943. They demonstrated a simplified computational model of how biological neurons might work together in their paper, “A Logical Calculus of Ideas Immanent in Nervous Activity” [33] They explained how the neurons use propositional logic to perform complex calculations in animal brains. All of this was nothing but the first architecture of

artificial neural networks. Many other types of architecture would be designed in the following years. Until the 1960s, there was a broad belief that humans would be able to interact with very intelligent machines, led by early successes of Artificial Neural Networks. The early 1980s saw an invention of new network architectures. Researchers were also developing better training techniques. This caused a revival of interest in ANNs. During the 1990s, the majority of researchers were favoring alternative Machine Learning techniques that could be more powerful, such as Support Vector Machines. Better results and stronger theoretical foundations were believed to be offered by these techniques. Nowadays the interest in ANNs is fortunately awakened again. [32]

3.3 Logical Computations with Neurons

The researchers mentioned above first came up with a very straightforward model of the biological neuron. This came to be known as an artificial neuron. The artificial neuron contains more than one inputs and only one output (usually both are binary). The output is easily activated when the active inputs surpass a certain number. McCulloch and Pitts presented how it can be conceivable to build artificial neurons into a network and compute any logical result you want, even using such a simplified model. In the following figure there are portrayed some ANNs that perform various logical computations, all while considering that the activation of a neuron happens only when at least two of its inputs are turned on (active).

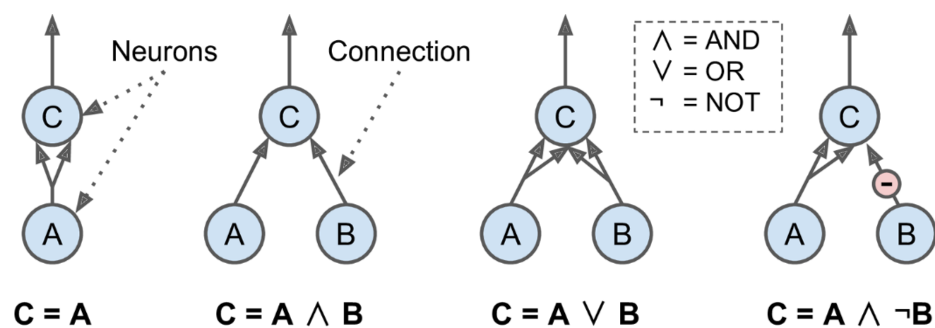


Figure 4. Simple Neurons Illustrated as Logic Gates [32]

One of the simplest ANN architectures is called a Perceptron. With numbers serving as inputs and outputs, this neuron is called a Linear Threshold Unit (LTU). The input is each on their own connected to a value, which is called weight. The Linear Threshold

Unit using the following formula calculates a weighted sum: ($z = w1\ x1 + w2\ x2 + \cdots + wn\ xn = wT \cdot x$). Afterwards a step method is applied to the sum computed previously. Finally, the product is finally outputted: $hw(x) = step\ (z) = step\ (wT \cdot x)$.

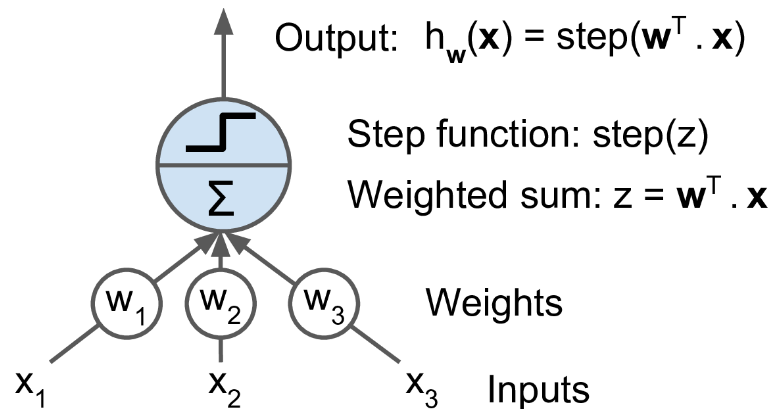


Figure 5. Anatomy of the Perceptron [32]

Multiple Perceptrons stacked together along with some bias values create what we call a Multi-Layered Perceptron, which is the core foundation of Deep Neural Networks used nowadays. A Multi=Layered Perceptron consists of one input layer, one or more middle layers that are hidden and one final output layer. The steps for each training instance go as follows: firstly, the backpropagation algorithm makes a prediction that is called forward passing. It measures the error contribution from each connection while going through each layer in reverse (reverse pass). Lastly, the connection weights are slightly tweaked in order to have the error reduced. The process of tweaking is called Gradient Descent step.
Instead of the logistic function, we may use the backpropagation algorithm together with other activation functions. Some other famous activation functions are:

1. *The hyperbolic tangent function tanh (z) = 2σ (2z) – 1*
   It is S-shaped, continuous, and differentiable, just like the logistic function. Unlike the logistic function, it has an output value that ranges from –1 to 1. This range contributes to causing each layer's output to be almost normalized, in other words: centered on zero at the start of training. All of this leads to a speedup of the convergence. [32]

2. *The ReLU function*

   ReLU (z) = max (0, z). Although continuous, this function can sadly not be differentiable at the point z = 0 (Gradient Descent bounces around since the slope changes abruptly). It is, however, quick to be calculated. This function also does not have a maximum output value and this can help reduce some issues during Gradient Descent [32]

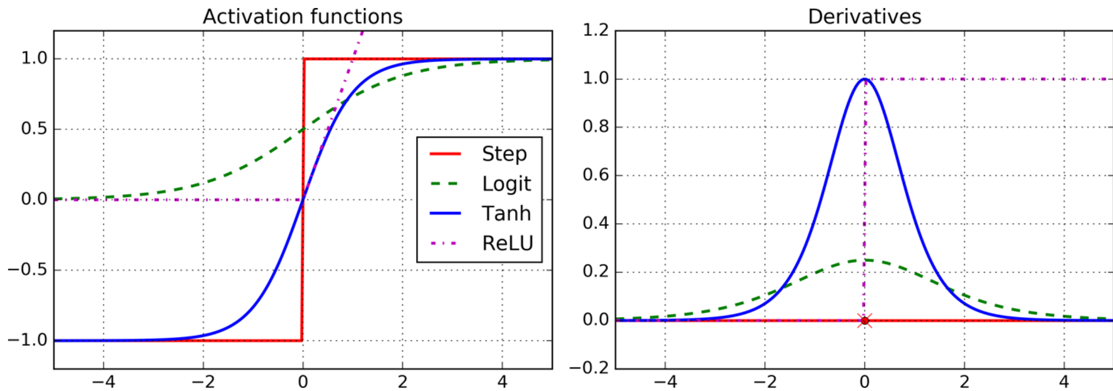


Figure 6. Activation Functions & their Derivatives

3.4 Computer Vision Techniques

Computer vision can be hard to define because it sits at the joint of several research and development fields, such as computer science (algorithms, data processing, and graphics), physics (optics and sensors), mathematics (calculus and information theory), and biology (visual and neural processing). At its core, computer vision can be summarized as the automated extraction of information from digital images. [32] A central goal in computer vision is to make sense of images, that is, to extract meaningful, semantic information from pixels (such as the objects present in images, their location, and their number). This generic problem can be divided into several sub-domains:

1. Object classification
2. Object identification
3. Object detection and localization
4. Object and instance segmentation

In computer vision, a **feature** is a piece of information (often mathematically represented as a one or two-dimensional vector) that is extracted from data that is relevant to the task. Features include some key points in the images, specific edges,

discriminative patches, and so on. They should be easy to obtain from new images and contain the necessary information for further recognition. Our images are complex structures with a large number of values (that is, H × W × D values with H indicating the image's height, W its width, and D its depth/number of channels, such as D = 3 for RGB images). This number of parameters simply explodes when we consider larger RGB images or deeper networks. Because their neurons receive all the values from the previous layer without any distinction (they are fully connected), these neural networks do not have a notion of distance/spatiality. More precisely, this means that the notion of proximity between pixels is lost to fully connected (FC) layers, as all pixel values are combined by the layers with no regard for their original positions. It is common practice to flatten multidimensional inputs before passing them to these layers. CNNs can handle multidimensional data. For images, a CNN takes as input three-dimensional data (height × width × depth) and has its own neurons arranged in a similar volume. This leads to the second novelty of CNNs—unlike fully connected networks, where neurons are connected to all elements from the previous layer, each neuron in CNNs only has access to some elements in the neighboring region of the previous layer. This region (usually square and spanning all channels) is called the **receptive field** of the neurons (or the filter size) [32]

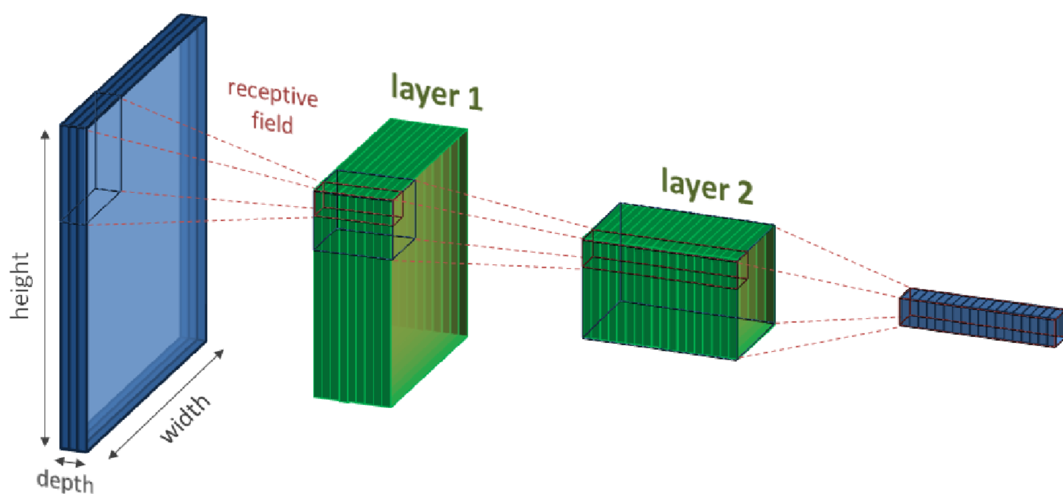


Figure 7. Simple CNN Architecture

CHAPTER 4

4. METHODOLOGY

4.1 VGG – a standard CNN architecture

VGG (or *VGGNet*), developed by the *Visual Geometry Group* from Oxford University, though only achieved second place in the ILSVRC classification task in 2014, influenced many later architectures. AlexNet was the first CNN successfully trained for such a complex recognition task and making several contributions that are still valid nowadays, such as:

The use of a **rectified linear unit** (***ReLU***) as an activation function prevents the vanishing gradient problem, and thus improving training (compared to using sigmoid or tanh). Also the application of **dropout** to CNNs. The typical CNN architecture combining blocks of convolution and pooling layers, with dense layers afterward for the final prediction. The application of random transformations (image translation, horizontal flipping, and more) to artificially augment the dataset (that is, augmenting the number of different training images by randomly editing the original samples. The main motivation of many researchers was to try going deeper (that is, building a network composed of a larger number of stacked layers), despite the challenges arising from this. More layers typically means more parameters to train, making the learning process more complex. Karen Simonyan and Andrew Zisserman from Oxford's VGG group tackled this challenge with success. [34] The method they submitted to ILSVRC 2014 reached a top-5 error of 7.3%, dividing the 16.4% error of AlexNet by more than two. In their paper (Simonyan, 2014) [34] presented how they developed their network to be deeper than most previous ones. They actually introduced six different CNN architectures, from 11 to 25 layers deep. Each network is composed of five blocks of several consecutive convolutions followed by a max-pooling layer and three final dense layers (with dropout for training). All the convolutional and max-pooling layers have SAME for padding. The convolutions have s = 1 for stride, and are using the ReLU function for activation. Overall, a typical VGG network is represented in the following diagram:

Figure 8. VGG-16 Architecture

The two most performant architectures, still commonly used nowadays, are called VGG-16 and VGG-19. The numbers 16 and 19 represent the depth of these CNN architectures; that is, the number of trainable layers stacked together. For example, as shown in the figure above, VGG-16 contains 13 convolutional layers and 3 dense ones, hence a depth of 16 (not including the non-trainable procedures; that is, the five max pooling and two dropout layers). The same goes for VGG-19, which is composed of three additional convolutions. VGG-16 has approx. 138 million parameters, and VGG-19 has 144 million.

The VGG authors then decided to replace the large convolutions with multiple smaller ones. A simple observation made by them was that a stack of two convolutions with 3 by 3 kernels and the same receptive field as a convolution with 5 by 5 kernels. Similarly, three consecutive convolutions of those 3 by 3 kernels resulted in a 7-by-7 receptive field and five convolutions resulted in an 11-by-11 receptive field. Therefore, while AlexNet had large filters up to 11 by 11, the VGG network contains more numerous but smaller convolutions for a larger efficient receptive field. There are two main benefits achieved from this observation. It decreases the number of parameters: The N filters of an 11 by 11 convolution layer imply 11 by 11 by $D \times N = 121DN$ values to train just for their kernels (for an input of depth D). While five 3 by 3 convolutions have a total of $1 \times (3 \times 3 \times D \times N) + 4 \times (3 \times 3 \times N \times N) = 9DN + 36N2$ weights for their kernels. As long as $N < 3.6D$, this means fewer parameters. For instance, for $N = 2D$, the number of parameters drops from 242D2 to 153D2. This makes the network easier to optimize, as well as much lighter. It also increases the non-linearity: Having a larger number of convolution layers, each followed by a non-linear activation function such as ReLU, increases the networks' capacity to learn complex features (that is, by combining more non-linear operations). (Simonyan, 2014) Also introduced a data augmentation mechanism that they named **scale jittering**. At each training iteration, they randomly scale the batched images (from 256 pixels to 512 pixels for their smaller

side) before cropping them to the proper input size (224 × 224 for their ILSVRC submission). With this random transformation, the network will be confronted with samples with different scales and will learn to properly classify them despite this scale jittering. The network becomes more robust as a result, as it is trained on images covering a larger range of realistic transformations. [32]

4.2 Faster R-CNN – a powerful object detection model

The Faster R-CNN architecture [1] was engineered over several years of research. More precisely, it was built incrementally from two architectures—R-CNN and Fast R-CNN. Faster R-CNN works in two stages:

1. The first stage is to extract a region of interest (RoI, or RoIs in the plural form). A RoI is an area of the input image that may contain an object. For each image, the first step generates about 2,000 RoIs.
2. The second stage is the classification step (sometimes referred to as the detection step). We resize each of the 2,000 RoIs to a square to fit the input of a convolutional network. We then use the CNN to classify the RoI. [1]

4.2.1 Region Proposals Network (RPN)

Regions of interest are generated using the **region proposal network** (**RPN**). To generate RoIs, the RPN uses convolutional layers. Therefore, it can be implemented on the GPU and is very fast. It uses anchor boxes—in the Faster R-CNN paper, nine anchor sizes are used (three vertical rectangles, three horizontal rectangles, and three squares). It can use any backbone to generate the feature volume and it uses a grid, and the size of the grid depends on the size of the feature volume. The network's last layer outputs numbers that allow the anchor box to be refined into a proper bounding box fitting the object. The RPN accepts an image as input and outputs regions of interest. Each region of interest consists of a bounding box and an objectness probability. To generate those numbers, a CNN is used to extract a feature volume. The feature volume is then used to generate the regions, coordinates, and probabilities.

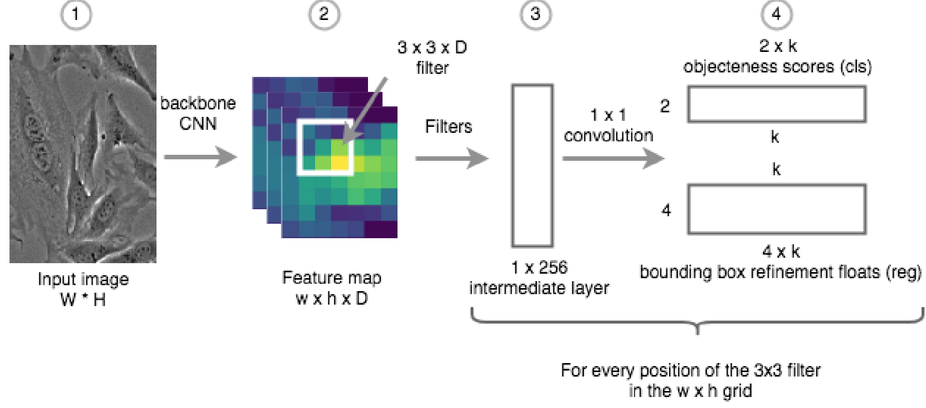


Figure 9. CNN Feature Map Flow

The systematic process represented in the figure above is as follows:

1. The network accepts an image as input and applies several convolutional layers.
2. It outputs a feature volume. A convolutional filter is applied over the feature volume. Its size is $3 \times 3 \times D$, where D is the depth of the feature volume.
3. At each position in the feature volume, the filter generates an intermediate $1 \times D$ vector.
4. Two sibling 1×1 convolutional layers compute the 'objectness' scores and the bounding box coordinates. There are two 'objectness' scores for each of the k bounding boxes. There are also four floats that will be used to refine the coordinates of the anchor boxes.

After post-processing, the final output is a list of RoIs. At this step, no information about the class of the object is generated, only about its location.

4.2.2 Mean Average Precision Score

Average precision gives information about the performance of a model for a single class. To get a global score, we use mean Average Precision (mAP) [35]. This corresponds to the mean of the average precision for each class. mAP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$AP = \sum_{n} (R_n - R_{n-1}) P_n$$

Where P_n and R_n are the precision and recall at the nth threshold. This implementation is not interpolated and is different from computing the area under the precision-recall curve with the trapezoidal rule, which uses linear interpolation and can be too optimistic. [35] The Mean average precision score was used to clearly evaluate the Faster-RCNN architecture through each model's tests. This score was generated by running the model against pre-labelled images and calculating the average precision of each image.

4.2.3 Classification inside Faster-RCNN

The second part of Faster R-CNN is the classification part. It outputs the final bounding boxes and accepts two inputs—the list of RoIs from the previous step (RPN), and a feature volume computed from the input image.
Since most of the classification stage architecture comes from the previous paper, Fast R-CNN, it is sometimes referred to with the same name.
Therefore, Faster R-CNN can be regarded as a combination of RPN and Fast R-CNN.
The classification part can work with any feature volume corresponding to the input image. However, as feature maps have already been computed in the previous region-proposal step, they are simply reused here. This technique has two benefits: Sharing the weights: If we were to use a different CNN, we would have to store the weights for two backbones: one for the RPN, and one for the classification, and secondly, sharing the computation: For one input image, we only compute one feature volume instead of two. As this operation is the most expensive of the whole network, not having to run it twice allows for a consequent gain in computational performance. For each RoI, convolutional layers are applied to obtain class predictions and bounding box refinement information.

While convolutional networks can accept inputs of any size (as they use a sliding window over the image), the final fully connected layer (between steps 2 and 3) accepts a feature volume of a fixed size as an input. In addition, since region, proposals are of different sizes (a vertical rectangle for a person, a square for an apple...); this makes the final layer impossible to use as is.
To circumvent that, a technique was introduced in Fast R-CNN—region of interest

pooling (RoI pooling). This converts a variable-size area of the feature map into a fixed-size area. The resized feature area can then be passed to the final classification layers.

4.2.4 RoI Pooling

The goal of the RoI pooling layer is simple; to take a part of the activation map of variable size and convert it into a fixed size. The input activation map sub-window is of size $h \times w$. The target activation map is of size $H \times W$. RoI pooling works by dividing its input into a grid where each cell is of size $h/H \times w/W$. Let us use an example. If the input is of size $h \times w = 5 \times 4$, and the target activation map is of size $H \times W = 2 \times 2$, then each cell should be of size 2.5×2. Because we can only use integers, we will make some cells of size 3×2 and others of size 2×2. Then, we will take the maximum of each cell:

	A	B	C	D	E	F	G	H	I	J	K
1	0.16	1.00	0.26	0.11	0.14	0.90	0.06	0.15			
2	0.13	0.05	0.58	0.34	0.58	0.13	0.78	0.53			
3	0.89	0.35	0.38	0.65	0.01	0.56	0.97	0.06			
4	0.47	0.97	0.78	0.99	0.82	0.90	0.32	0.89		0.97	0.99
5	0.58	0.13	0.12	0.50	0.99	0.35	0.83	0.39		0.96	0.93
6	0.21	0.59	0.96	0.93	0.08	0.55	0.13	0.89			
7	0.03	0.83	0.63	0.46	0.09	0.03	0.68	0.13			
8	0.82	0.35	0.20	0.48	0.80	0.41	0.46	0.08			

Figure 10. RoI Max Pooling Example

A RoI pooling layer is very similar to a max pooling layer. The difference is that RoI pooling works with inputs of variable size, while max-pooling works with a fixed size only. RoI pooling is sometimes referred to as RoI max pooling. In the original R-CNN paper, RoI pooling had not yet been introduced. Therefore, each RoI was extracted from the original image, resized, and directly passed to the convolutional network. Since there were around 2,000 RoIs, it was extremely slow. The Fast in Fast RCNN comes from the huge speedup introduced by the RoI pooling layer. [32]

4.2.5 Intersection over Union Threshold (IOU)

The number of predictions matching or not matching the ground truth boxes defines true and false positives. In order to decide when a prediction and the ground truth are matching a common metric is used: *the Jaccard index*, which measures how well two sets overlap (in our case, the sets of pixels represented by the boxes). Also known as Intersection over Union (IOU), it is defined as follows:

$$IoU(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A + B| - |A \cap B|}$$

$|A|$ and $|B|$ are the number of elements that each set contains. $A \cap B$ is the intersection of the two sets, and therefore the numerator $|A \cap B|$ represents the number of elements they have in common. Similarly, $A \cup B$ is the union of the sets and therefore the denominator $|A \cup B|$ represents the total number of elements the two sets cover together. [32]

While the intersection would provide a good indicator of how much two sets/boxes overlap, this value is absolute and not relative. Therefore, two big boxes would probably overlap by many more pixels than two small boxes. This is why this ratio is used—it will always be between 0 (if the two boxes do not overlap) and 1 (if two boxes overlap completely). When computing the average precision, we say that two boxes overlap if their IOU is above a certain threshold.

The threshold chosen for the experiments in this paper is 0.5 for the classifier and 0.7 for the RPN layer.

4.3 Labelling the Dataset

In the initial part of the experiments a type of labelling was required which was supported by the current Faster-RCNN architecture we have presented before. This labelling was done using an open source image annotation tool such as LabelImg. [36] This tool is used to label object bounding boxes in images. The tool is run via a python interface and it allows the user to select areas in a specific image defined as bounding boxes and assign a label to each bounding box. It is written in Python and uses Qt for its graphical interface. Annotations are saved as XML files in PASCAL VOC format, the format used by ImageNet. [37]

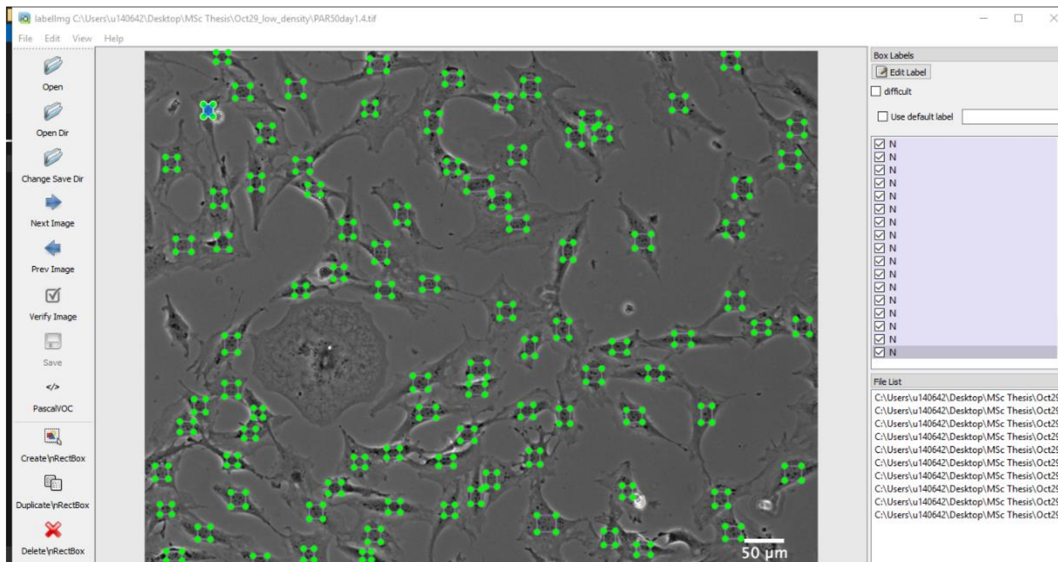


Figure 11. LabelImg User Interface

For the label, the letter 'N' was used in each bounding box to represent the Nuclei of the cells. The xml file format in which labelimg saves the files is not supported by default by our Faster-RCNN architecture so a conversion was needed. A predefined python script 'xml_to_csv.py' [8.1] was used for the conversion of the labels from xml to 'comma separated value' format. This format was suitable for the architecture used and is the final step in labelling. Labelling was done in two parts. The first part consisted of 46 images labelled which were used for the two first models trained. 30 images were used for training and 16 for testing. Later the dataset was doubled by adding 40 more images and splitting them into 60 images for training and 26 for testing. This part of the research required the most manual work with over 100 hours of labelling spread over 3 months.

CHAPTER 5

5. RESULTS AND DISCUSSIONS

5.1 Initial Model 03

VGG-16 Hyper-parameters:

During the initial testing, a dataset of 44 images was used. This was split into 30 images for training the VGG-16 network and 14 other images for testing validation. In total, there were 2752 nuclei labels in the training dataset. The number of classes to distinguish between was set to two; one being the 'N' (nucleus) and the other was automatically set to 'bg' (background). The original images had a resolution of 1280x1024 and this was not resized during training or testing. The images were kept in greyscale to provide higher efficiency against RGB. The feature map size was with a height=64 and width=80 whereas the RPN stride was 16 pixels.

VGG-16 Initial Results:

Firstly, Anchor Box Scales of [64, 128, 256] were tested against this dataset. The training of 115k Batches took over 30 hours on Google Colab. We can see the results of Batch 115k in the tables below:

Table 1. Model 03 Results

Mean Overlapping Bounding boxes	***Classifier accuracy***	***RPN Classifier Loss***	***RPN Regression Loss***
24.262	0.797	0.126	0.103

Classifier Classification Loss	***Classifier Regression Loss***	***Current Loss***	***Elapsed time***
0.415	0.091	0.735	32.27

The network was able to achieve **a mean average precision of 72.8 %** in terms of accuracy.

In this model, one of the main challenges we had to improve was the low number of detections. This was coming due to the network scanning for larger anchor boxes when training the Regions Proposal Network. The anchor boxes were not in proportion to the mean size of a nucleus and this way the network was detecting only those nucleus, which were grouped together due to the larger window scanning for these targets. In the next model, we decided to improve on that anchor box scale and train the model again.

Some of the results from model_03 look like the following images:

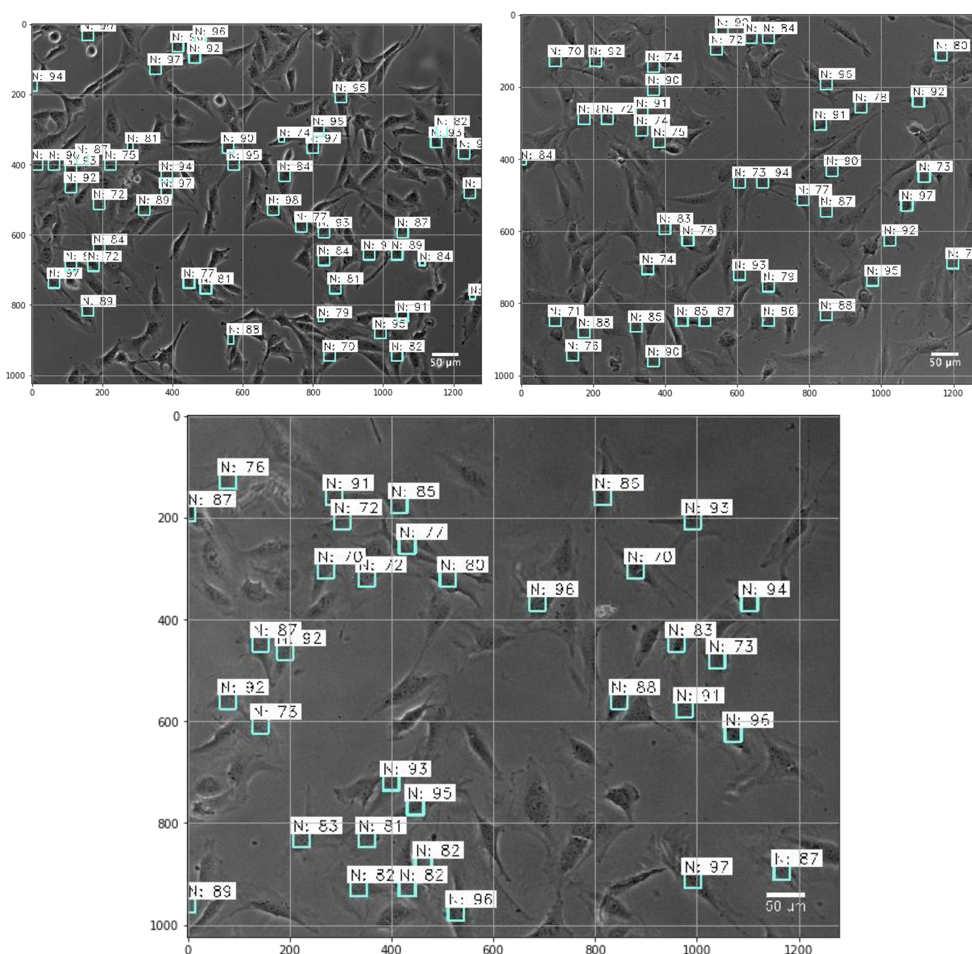
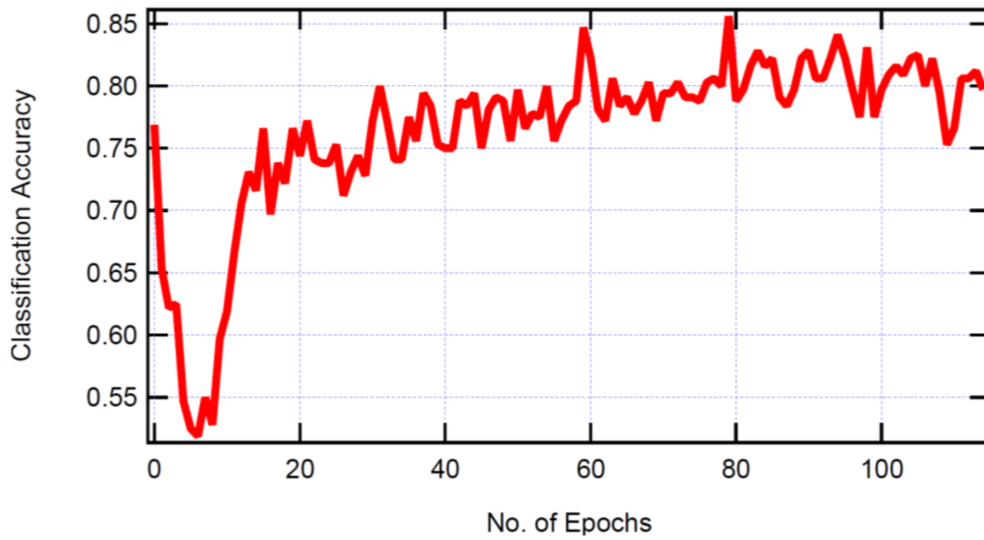
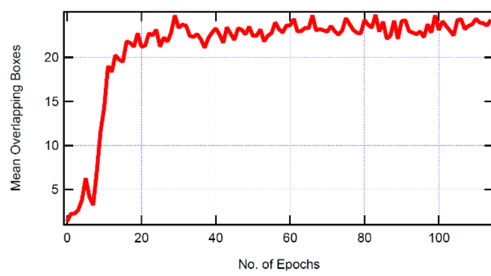
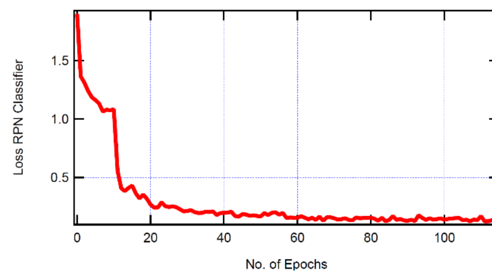


Figure 12. Model 03 Test Results



M03 Classification Accuracy





M03 Mean Overlapping Boxes

M03 Loss RPN Classification

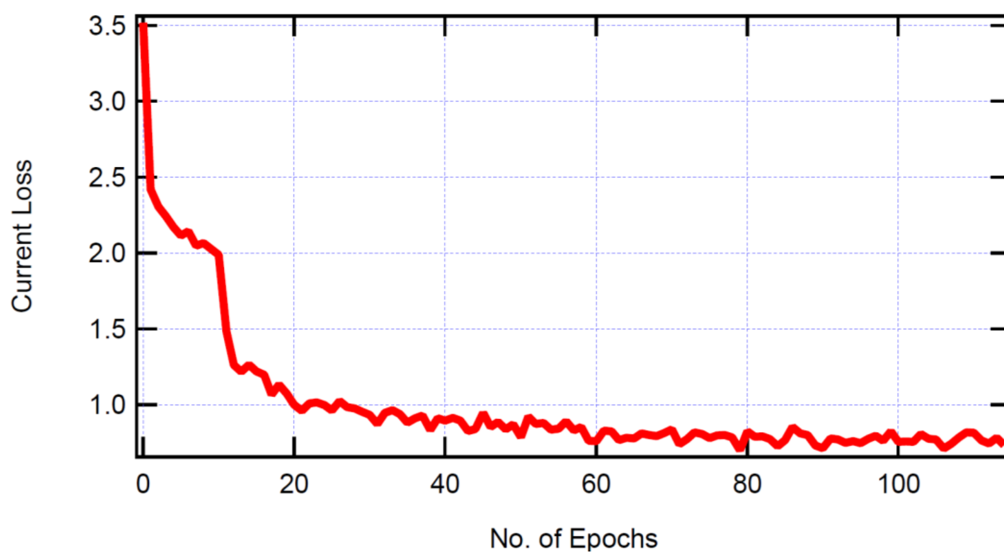


Figure 13. M03 Total Loss & Other Graphs (above)

5.2 Improvements in Model 04

Later the Anchor Box Scales were lowered to [32, 64, 128] in order to fit the cell size better. 85k Batches were trained for over 28 hours on Google Colab. As expected, we noticed changes in the results for the 85k Batch:

Table 2. Model 04 Results

Mean Overlapping Bounding boxes	***Classifier accuracy***	***RPN Classifier Loss***	***RPN Regression Loss***
26.983	0.829	0.16	0.13

Classifier Classification Loss	***Classifier Regression Loss***	***Current Loss***	***Elapsed time***
0.375	0.046	0.71	32.27

The lower scales led to a higher **mean average precision of 82.6%**, an increase by approximately 10%.

However the images of testing suffered from this precision increase in terms of cells being more exclusively distinguished by the VGG-16.

The increasing accuracy was a hopeful achievement but the problem of low detection persisted in model_04. At this phase, we decided to improve upon the preprocessing of the images and trained model_05 based on this. Some results of model_04 are listed below:

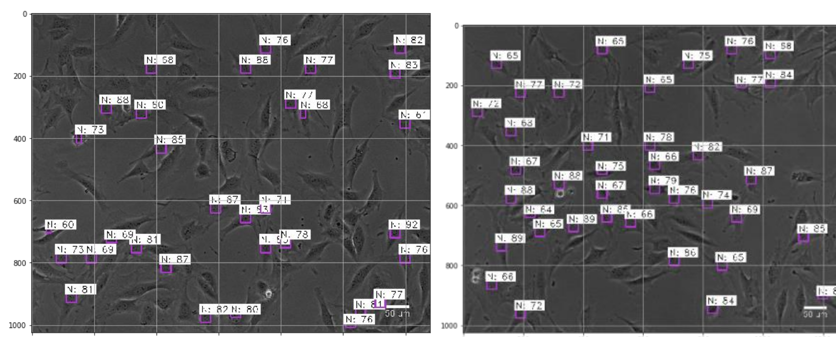
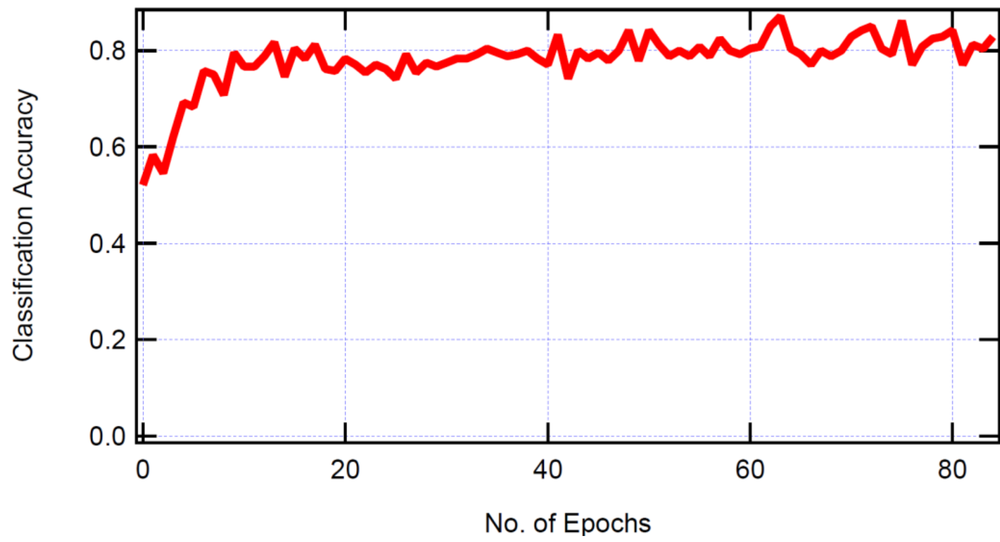
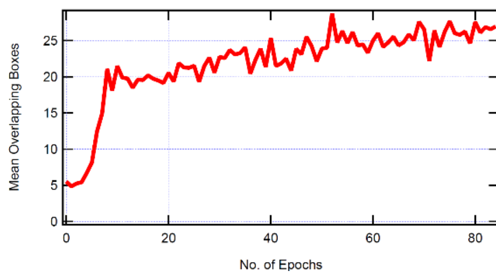
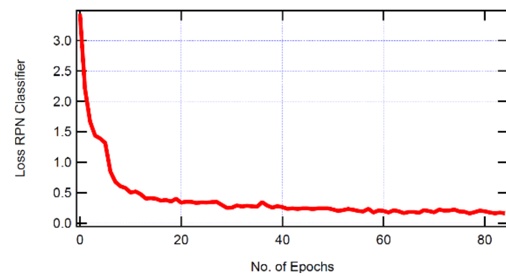


Figure 14. Model 04 Test Results



M04 Classification Accuracy





M04 Mean Overlapping Boxes M04 Loss RPN Classification

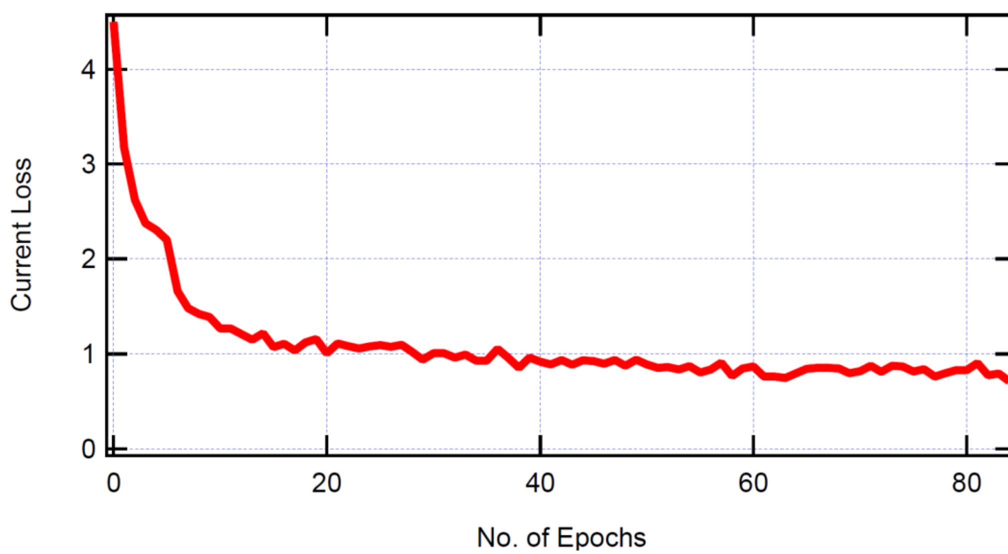


Figure 15. M04 Total Loss & other graphs (above)

5.3 Histogram Matching Preprocessing in Model 05

Afterwards, the histogram matching technique was applied to the dataset and its size was increased by doubling the number of labelled images of cells. As we can see from the image below, there is a huge contrast difference between the upper left corner of the image and the brighter lower right corner. This is because most images are not taken in an ideal environment and the lighting/focus elements are disregarded for our dataset.

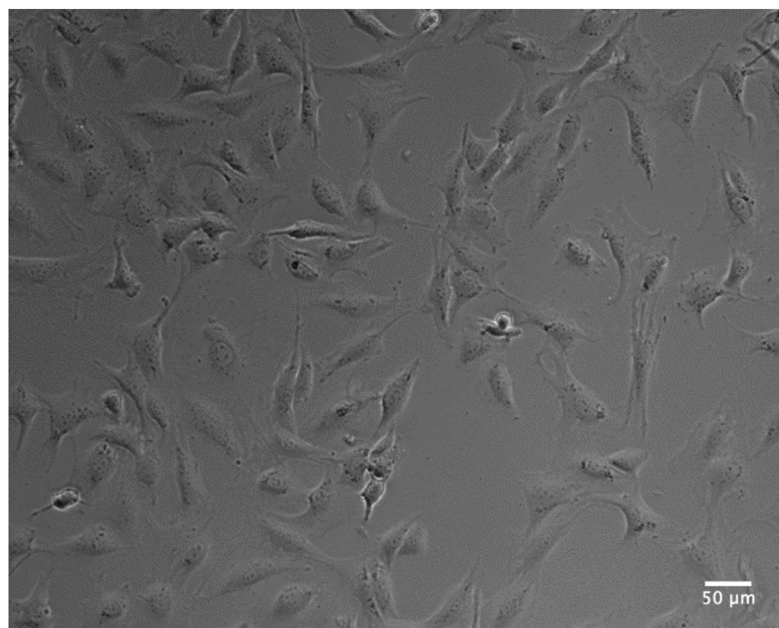


Figure 16. Difference of contrast inside a sample

For this reason, we applied a template matching technique used widely in image processing to correct this issue and introduce a uniform set of pixel intensities for the neural network to process. This technique consists of an image being selected as a template to be matched and all the other images in the dataset have their histograms adjusted in order to 'match' the template image intensities. One example of the process is illustrated in the figure below. The histogram matching python code is provided in the appendix section of this paper.

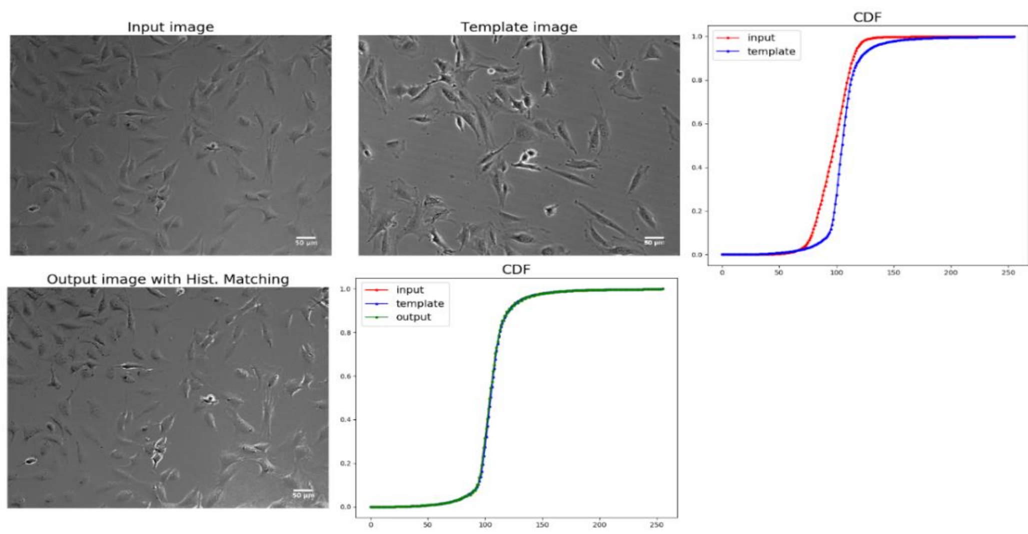


Figure 17. Histogram Matching Technique

This technique makes use of the cumulative distribution function of the intensities of each pixel and based on this information it applies the template to the input to generate a specific output. Our model then evaluates this output and since it has similar group intensities, the resulting images with detected bounding boxes were easier for the network to specify which regions can be nuclei and which are background pixels.

The resulting images with cells detected were far more promising than the previous tests:

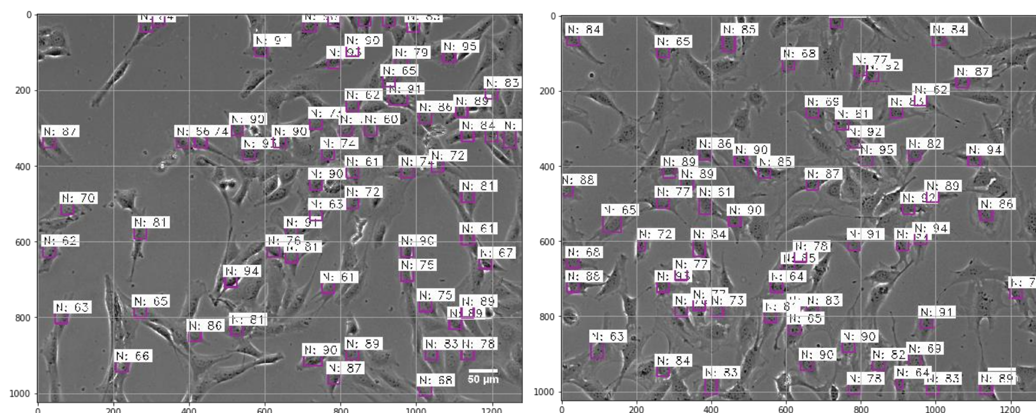
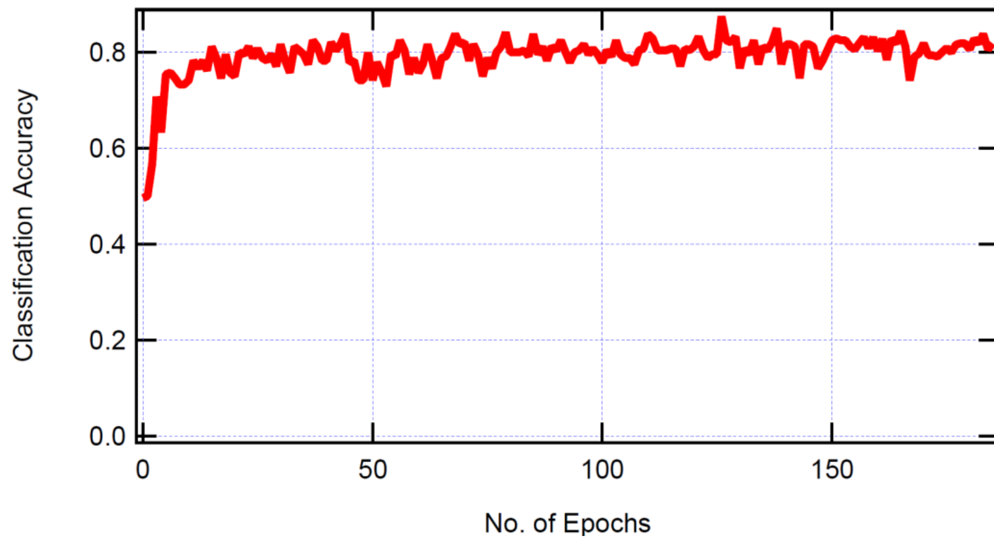
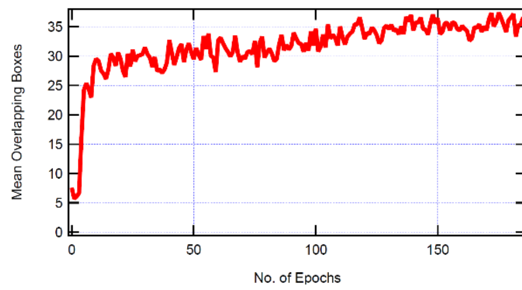


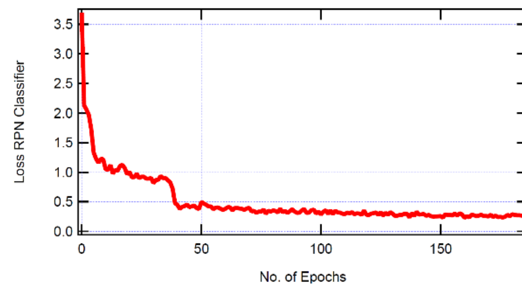
Figure 18. Model 05 Test Results



M05 Classification Accuracy



M05 Mean Overlapping Boxes



M05 Loss RPN Classification

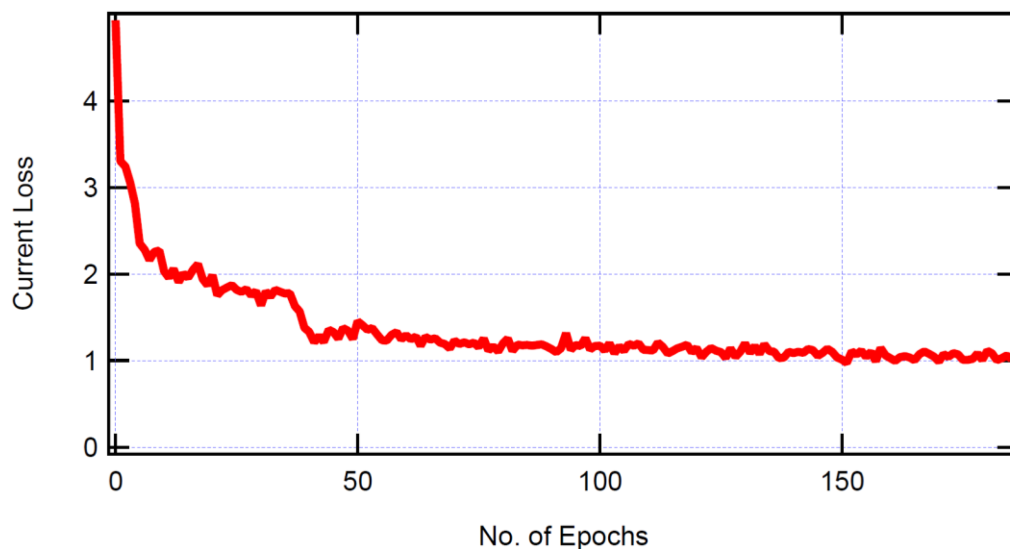


Figure 19. M05 Total Loss & other graphs (above)

In this model_05, 187 Epochs were trained with over 60 hours of enhanced cloud computing on Google Colab. The following numerical results were reached on batch 187.000:

Table 3. Model 05 Results

Mean Overlapping Bounding boxes	**Classifier accuracy**	**RPN Classifier Loss**	**RPN Regression Loss**
36.638	0.819	0.254	0.186

Classifier Classification Loss	**Classifier Regression Loss**	**Current Loss**
0.391	0.192	1.024

After training 187k batches, the **mean average precision reached was 79.21%**. A lower value than what we had before but still justified by the fact that the dataset is doubled now and thus the training length requirements are extended to reach the same level of mAP. As the next step an improvement of the histogram matching was proposed by deepening the classification of images before conducting histogram matching.

Even after such a good resulting model, we knew there was stillroom for improvement. That is when we noticed some of the images developing sort of ‘auerolas’ the cells when template matching was used. These bright spots surrounding the cells were a result of an increase in contrast in a specific area of the image, which was already bright enough. That posed a problem for our model since it might throw the network off in not detecting any cells since the area is made of high intensity pixels. For this issue we decided to group the images further before we continued with histogram matching. The following results of model_06 describe this method in detail.

5.4 Selective-Histogram Matching Classification in Model 06

In this part, a method of deepening the manual classification of images was proposed. This technique consisted in splitting the dataset into three subgroups depending on their contrast level. Respectively High-, Medium- and Low-Contrast groups were created and images were manually classified into them based on the level of contrast of each image.

An example is demonstrated below with two images of High-Contrast (left) and Low-Contrast (right) before and after the technique of "selective histogram matching".

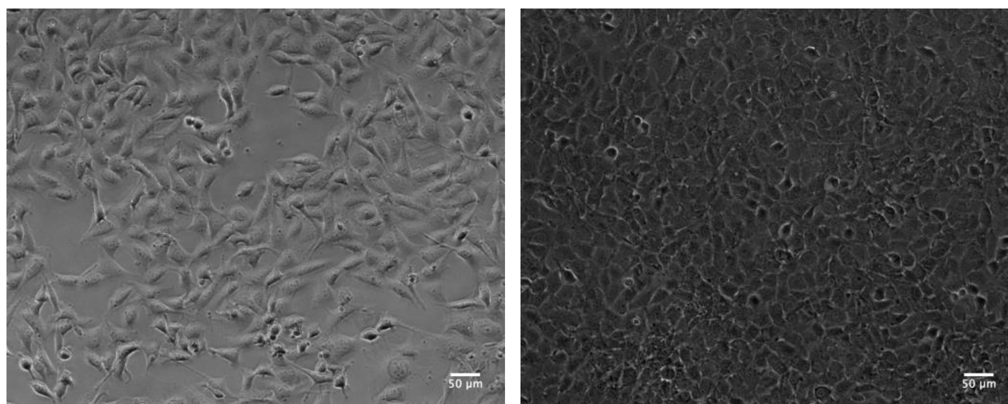


Figure 20. Images before Selective Histogram Matching

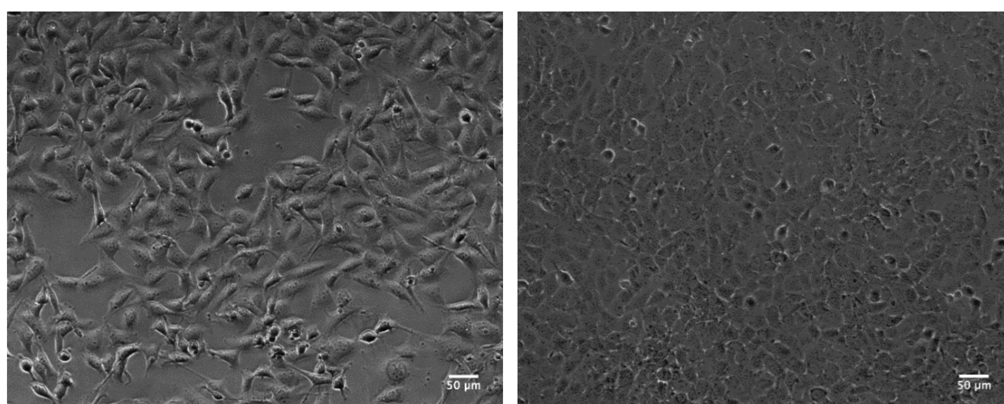
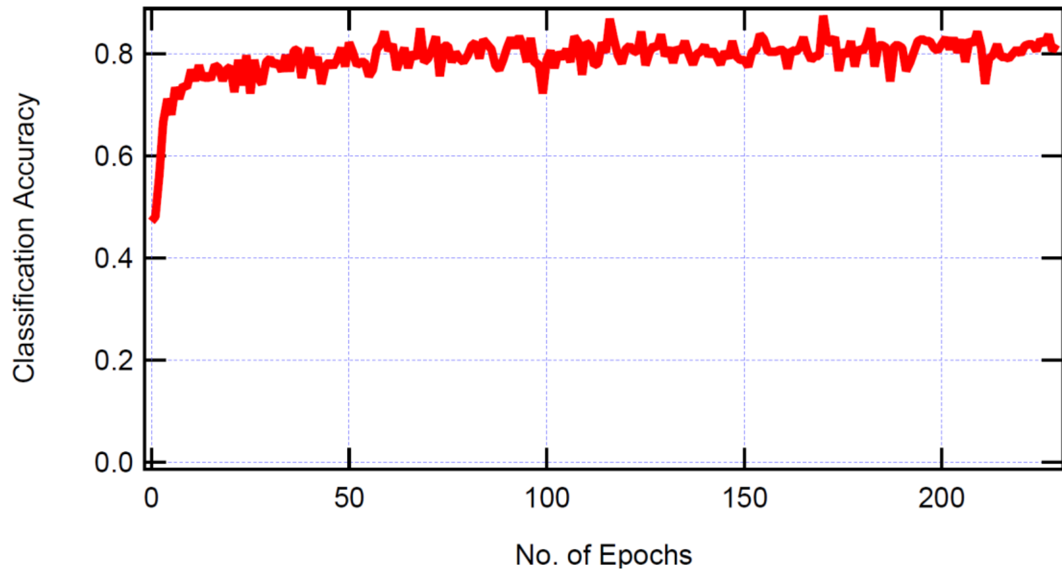
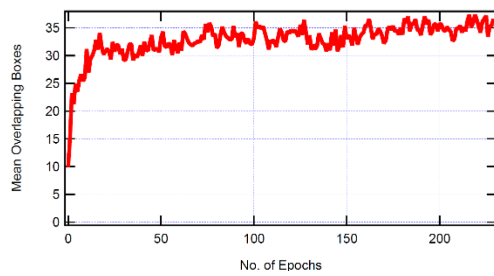


Figure 21. Images after Selective Histogram Matching

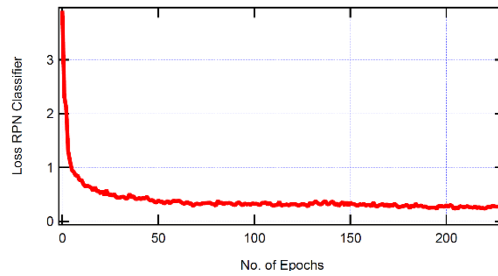
We can surely notice how the high contrast image had its contrast tuned down and turned a bit darker, while the opposite effect is apparent in the low contrast image. This grouping of images before applying histogram matching was made in order to reduce differentiation between images in times of training the model but also to yield better results during test runs.



M06 Classification Accuracy



M06 Mean Overlapping Boxes



M06 Loss RPN Classification

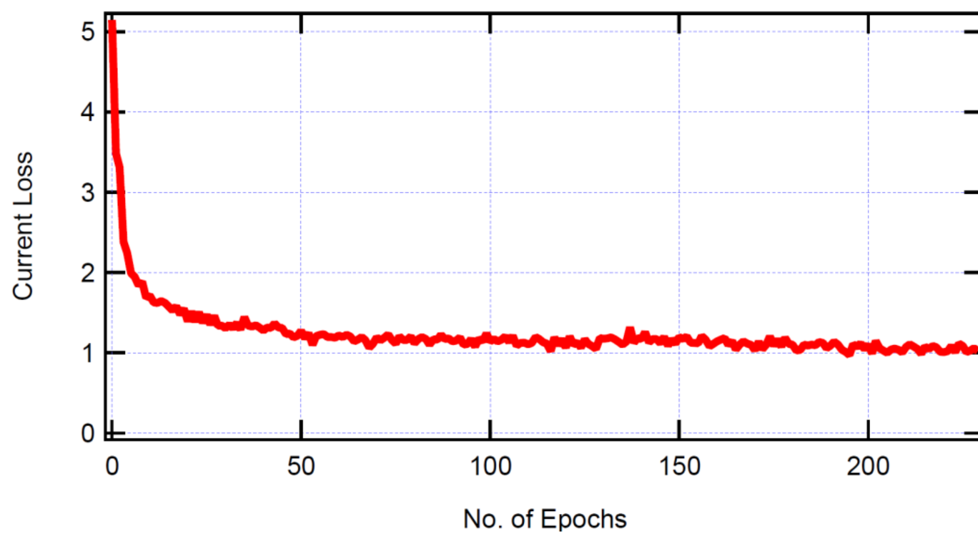


Figure 22. M06 Total Loss & other graphs (above)

In this final and best model so far, 130 Epochs were trained with over 40 hours of enhanced cloud computing on Google Colab. The following numerical results were reached on batch 130.000:

Table 4. *Model 06 Results*

Mean Overlapping Bounding boxes	Classifier accuracy	RPN Classifier Loss	RPN Regression Loss
34.3	0.812	0.272	0.193

Classifier Classification Loss	Classifier Regression Loss	Current Loss
0.414	0.168	1.047

After training 130k batches, the **mean average precision reached was 78.07%**. This demonstrates a trend of declining mean average precision as we get better and better results in the number of cells detected. This could be due to the fewer number of cells detected could be correlated with a lesser division for the mean average denominator and thus producing slightly higher values for the accuracy when we have few cells detected.

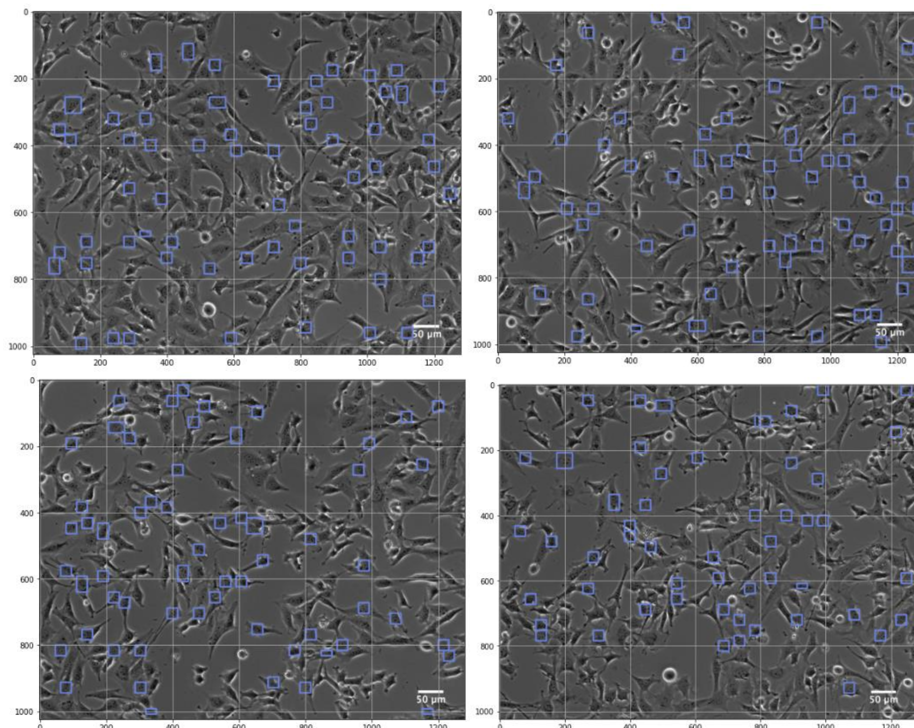


Figure 23. M06 Test Results

CHAPTER 6

6. CONCLUSIONS

6.1 Conclusions

Through these experiments we have concluded that lowering the bounding box scale for the FRCNN will adjust the network to achieve better accuracy in detecting small crowded cells in an image. The dataset requires better preprocessing such as the template matching technique used in digital image processing. This technique is a great improvement but still introduces some areas with higher threshold of intensity, which can throw the network off in defining the cells to be counted.

A huge improvement on the number of cells detected was the template matching technique applied in groups. Instead of a single template for all the dataset, the image was clustered into three groups of high, medium and low contrast and histogram-matching technique was applied for each respective group separately. This way a better distribution of pixel intensities was provided while keeping the image data as close as possible to the raw data. The model was also run against raw unlabeled images and resulted in very good detection numbers. Although one thing to be notices was the number of cells detected was always in an average of 50-70 cells. This could be a bias of the model based on the low density of the dataset.

6.2 Future Work

As future work, it is proposed that the current model can immeasurably help in labeling images by utilizing the bounding box coordinates detected by the model. This way the model can self-improve by minimizing the manual work required for labelling future images. In addition, the histogram matching technique still requires some sort of human labor to distinguish and classify the images in the similar groups of contrasts. A model can be created to automatically classify the similar histogram distribution, contrast of the images, and group them together. Then histogram matching can be applied, thus compressing the entire preprocessing stage into one simple script. Finally, the model can be further analyzed by fine-tuning the current model 06 with a wider dataset of higher density labelled images of cells.

7. REFERENCES

[1] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," Jun. 2015, [Online]. Available: http://arxiv.org/abs/1506.01497.

[2] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation BT - Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015," 2015, pp. 234–241.

[3] C. Dollinger *et al.*, "Controlling Incoming Macrophages to Implants: Responsiveness of Macrophages to Gelatin Micropatterns under M1/M2 Phenotype Defining Biochemical Stimulations," *Adv. Biosyst.*, vol. 1, no. 6, p. 1700041, Jun. 2017, doi: 10.1002/adbi.201700041.

[4] X. Polisi, A. Halili, C.-E. Tanase, A. Uka, N. E. Vrana, and A. Ghaemmaghami, "Computer Assisted Analysis of the Hepatic Spheroid Formation BT - Computational Bioengineering and Bioinformatics," 2020, pp. 117–126.

[5] A. Uka, X. Polisi, A. Halili, C. Dollinger, and N. E. Vrana, "Analysis of cell behavior on micropatterned surfaces by image processing algorithms," in *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, 2017, pp. 75–78, doi: 10.1109/EUROCON.2017.8011080.

[6] C. X. Hernández, M. M. Sultan, and V. S. Pande, "Using Deep Learning for Segmentation and Counting within Microscopy Data," Feb. 2018, [Online]. Available: http://arxiv.org/abs/1802.10548.

[7] I. Suleymanova *et al.*, "A deep convolutional neural network approach for astrocyte detection," *Sci. Rep.*, vol. 8, no. 1, p. 12878, Dec. 2018, doi: 10.1038/s41598-018-31284-x.

[8] J. Barthes *et al.*, "Controlling porous titanium/soft tissue interactions with an innovative surface chemical treatment: Responses of macrophages and fibroblasts," *Mater. Sci. Eng. C*, vol. 112, no. August, 2020, doi: 10.1016/j.msec.2020.110845.

[9] K. Sun *et al.*, "High-Resolution Representations for Labeling Pixels and Regions," Apr. 2019, [Online]. Available: http://arxiv.org/abs/1904.04514.

[10] J. F. Bonnefon, A. Shariff, and I. Rahwan, "The social dilemma of autonomous vehicles," *Science (80-.).*, vol. 352, no. 6293, pp. 1573–1576, 2016, doi:

10.1126/science.aaf2654.

[11] A. Rossler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Niessner, “FaceForensics++: Learning to detect manipulated facial images,” *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2019-Octob, pp. 1–11, 2019, doi: 10.1109/ICCV.2019.00009.

[12] Z. Luo, Y. Zhang, L. Zhou, B. Zhang, J. Luo, and H. Wu, “Micro-Vessel Image Segmentation Based on the AD-UNet Model,” *IEEE Access*, vol. 7, pp. 143402–143411, 2019, doi: 10.1109/ACCESS.2019.2945556.

[13] O. B. Hoque, M. I. Jubair, M. S. Islam, A. Akash, and A. S. Paulson, “Real Time Bangladeshi Sign Language Detection using Faster R-CNN,” in *2018 International Conference on Innovation in Engineering and Technology (ICIET)*, 2018, pp. 1–6, doi: 10.1109/CIET.2018.8660780.

[14] C. N. Vasconcelos and B. N. Vasconcelos, “Convolutional Neural Network Committees for Melanoma Classification with Classical And Expert Knowledge Based Image Transforms Data Augmentation,” Feb. 2017, [Online]. Available: http://arxiv.org/abs/1702.07025.

[15] D. Wang *et al.*, “AFP-Net: Realtime Anchor-Free Polyp Detection in Colonoscopy,” Sep. 2019, [Online]. Available: http://arxiv.org/abs/1909.02477.

[16] Z. Gao, L. Wang, L. Zhou, and J. Zhang, “HEp-2 Cell Image Classification With Deep Convolutional Neural Networks,” *IEEE J. Biomed. Heal. Informatics*, vol. 21, no. 2, pp. 416–428, 2017, doi: 10.1109/JBHI.2016.2526603.

[17] A. Kaku *et al.*, “DARTS: DenseUnet-based Automatic Rapid Tool for brain Segmentation,” Nov. 2019, [Online]. Available: http://arxiv.org/abs/1911.05567.

[18] X. Dong *et al.*, “Air, bone and soft-tissue Segmentation on 3D brain MRI Using Semantic Classification Random Forest with Auto-Context Model,” Nov. 2019, [Online]. Available: http://arxiv.org/abs/1911.09264.

[19] M. Frey and M. Nau, “Memory Efficient Brain Tumor Segmentation Using an Autoencoder-Regularized U-Net BT - Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries,” 2020, pp. 388–396.

[20] X. Chen and A. Gupta, “An Implementation of Faster RCNN with Study for Region Sampling,” Feb. 2017, [Online]. Available:

http://arxiv.org/abs/1702.02138.

[21] H. Tang, D. R. Kim, and X. Xie, "Automated pulmonary nodule detection using 3D deep convolutional neural networks," in *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*, 2018, pp. 523–526, doi: 10.1109/ISBI.2018.8363630.

[22] P. Doll, R. Girshick, and F. Ai, "Mask R-CNN ar."

[23] Z. Tang, K. Chen, M. Pan, M. Wang, and Z. Song, "An Augmentation Strategy for Medical Image Processing Based on Statistical Shape Model and 3D Thin Plate Spline for Deep Learning," *IEEE Access*, vol. 7, pp. 133111–133121, 2019, doi: 10.1109/ACCESS.2019.2941154.

[24] E. Carver, Z. Dai, E. Liang, J. Snyder, and N. Wen, "Improvement of Multiparametric MR Image Segmentation by Augmenting the Data with Generative Adversarial Networks for Glioma Patients," Oct. 2019, [Online]. Available: http://arxiv.org/abs/1910.00696.

[25] B. Seo *et al.*, "Cardiac MRI Image Segmentation for Left Ventricle and Right Ventricle using Deep Learning," Sep. 2019, [Online]. Available: http://arxiv.org/abs/1909.08028.

[26] C. Wang, Z. Zhao, Q. Ren, Y. Xu, and Y. Yu, "Dense U-net Based on Patch-Based Learning for Retinal Vessel Segmentation," *Entropy*, vol. 21, no. 2, p. 168, Feb. 2019, doi: 10.3390/e21020168.

[27] J. H. Tan, U. R. Acharya, S. V. Bhandary, K. C. Chua, and S. Sivaprasad, "Segmentation of optic disc, fovea and retinal vasculature using a single convolutional neural network," *J. Comput. Sci.*, vol. 20, pp. 70–79, May 2017, doi: 10.1016/j.jocs.2017.02.006.

[28] T. A. Soomro *et al.*, "Impact of Image Enhancement Technique on CNN Model for Retinal Blood Vessels Segmentation," *IEEE Access*, vol. 7, pp. 158183–158197, 2019, doi: 10.1109/ACCESS.2019.2950228.

[29] X. Feng, K. Qing, N. J. Tustison, C. H. Meyer, and Q. Chen, "Deep convolutional neural network for segmentation of thoracic organs-at-risk using cropped 3D images," *Med. Phys.*, vol. 46, no. 5, pp. 2169–2180, May 2019, doi: 10.1002/mp.13466.

[30] A. Duarte, C. V. Hegde, A. Kaku, S. Mohan, and J. G. Raya, "Knee Cartilage Segmentation Using Diffusion-Weighted MRI," Dec. 2019, [Online]. Available: http://arxiv.org/abs/1912.01838.

[31] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, May 2006, doi: 10.1162/neco.2006.18.7.1527.

[32] B. Planche and E. Andres, *Hands-On Computer Vision with TensorFlow 2*. 2019.

[33] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943, doi: 10.1007/BF02478259.

[34] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Sep. 2014, [Online]. Available: http://arxiv.org/abs/1409.1556.

[35] "sci-kit learn mAP." https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn-metrics-average-precision-score.

[36] "LabelImg." https://github.com/tzutalin/labelImg.

[37] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255, doi: 10.1109/CVPR.2009.5206848.

8. APPENDIX

8.1 Python Codes

xml_to_csv.py

```
import os
import glob
import pandas as pd
import xml.etree.ElementTree as ET


def xml_to_csv(path,folder):
    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            value = ('/content/drive/My Drive/test_colab01/'+folder+'/' +
root.find('filename').text,
                     # int(root.find('size')[0].text),
                     # int(root.find('size')[1].text),
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text),
                     member[0].text,
                     )
            xml_list.append(value)
    column_name = ['filename', 'xmin', 'ymin', 'xmax', 'ymax', 'class']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df


def main():
    for folder in ['train','test']:
    # folder = 'train'
        image_path = os.path.join(os.getcwd(), (folder))
        xml_df = xml_to_csv(image_path,folder)
        xml_df.to_csv((folder + '_labels.csv'), index=None)
        print('Successfully converted xml to csv.')


main()
```

histogram_matching.py

```
import numpy as np
from skimage.io import imread
from skimage.exposure import cumulative_distribution
from PIL import Image
import matplotlib.pylab as pylab
import os
from preproc import plot_image


def cdf(im):
    '''
    computes the CDF of an image im as 2D numpy ndarray
    '''
    c, b = cumulative_distribution(im)
    # pad the beginning and ending pixels and their CDF values
    c = np.insert(c, 0, [0]*b[0])
    c = np.append(c, [1]*(255-b[-1]))
    return c

def hist_matching(c, c_t, im):
    '''
    c: CDF of input image computed with the function cdf()
    c_t: CDF of template image computed with the function cdf()
    im: input image as 2D numpy ndarray
    returns the modified pixel values for the input image
    '''
    pixels = np.arange(256)
    # find closest pixel-matches corresponding to the CDF of the input image, given
the value of the CDF H of
    # the template image at the corresponding pixels, s.t. c_t = H(pixels) <=> pixels
= H-1(c_t)
    new_pixels = np.interp(c, c_t, pixels)
    im = (np.reshape(new_pixels[im.ravel()], im.shape)).astype(np.uint8)
    return im

def get_imlist(path):
  """  Returns a list of filenames for
    all jpg images in a directory. """
  return [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.tif')]

pylab.gray()

list = get_imlist('high')
im_t = ((imread('high/PAR50.50_92.tif'))).astype(np.uint8)
for i in range(0,len(list)):
    im = (imread(os.path.join(os.getcwd(), list[i]))).astype(np.uint8)
    c = cdf(im)
    c_t = cdf(im_t)
    im1 = hist_matching(c, c_t, im)
    image = Image.fromarray(im1)
    imsaved = image.save(list[i])
```

frcnn_train.py

```
#!/usr/bin/env python
# coding: utf-8

from google.colab import drive
drive.mount('/content/drive', force_remount=True)
#(10^3)*N_cell/(1024*1280)=cell density


get_ipython().run_line_magic('tensorflow_version', '1.x')


# ### Import libs


from __future__ import division
from __future__ import print_function
from __future__ import absolute_import
import random
import pprint
import sys
import time
import numpy as np
from optparse import OptionParser
import pickle
import math
import cv2
import copy
from matplotlib import pyplot as plt
import tensorflow as tf
import pandas as pd
import os
import tensorflow.keras as keras

from sklearn.metrics import average_precision_score

from keras import backend as K
from keras.optimizers import Adam, SGD, RMSprop
from keras.layers import Flatten, Dense, Input, Conv2D, MaxPooling2D, Dropout
from keras.layers import GlobalAveragePooling2D, GlobalMaxPooling2D, TimeDistributed
from keras.engine.topology import get_source_inputs
from keras.utils import layer_utils
from keras.utils.data_utils import get_file
from keras.objectives import categorical_crossentropy

from keras.models import Model
from keras.utils import generic_utils
from keras.engine import Layer, InputSpec
from keras import initializers, regularizers

print(tf.__version__)

# #### Config setting

get_ipython().system('ls')

class Config:

     def __init__(self):

           # Print the process or not
           self.verbose = True

           # Name of base network
           self.network = 'vgg'
```

```
        # Setting for data augmentation
        self.use_horizontal_flips = False
        self.use_vertical_flips = False
        self.rot_90 = False

        # Anchor box scales
    # Note that if im_size is smaller, anchor_box_scales should be scaled
    # Original anchor_box_scales in the paper is [128, 256, 512]
        # self.anchor_box_scales = [64, 128, 256]
        #self.anchor_box_scales = [24, 36, 64]
        self.anchor_box_scales = [32, 64, 128]

        # Anchor box ratios
        self.anchor_box_ratios = [[1, 1], [1./math.sqrt(2), 2./math.sqrt(2)],
[2./math.sqrt(2), 1./math.sqrt(2)]]

        # Size to resize the smallest side of the image
        # Original setting in paper is 600. Set to 300 in here to save training
time
        self.im_size = 1024

        # image channel-wise mean to subtract
        self.img_channel_mean = [103.939, 116.779, 123.68]
        self.img_scaling_factor = 1.0

        # number of ROIs at once
        self.num_rois = 4

        # stride at the RPN (this depends on the network configuration)
        self.rpn_stride = 16

        self.balanced_classes = False

        # scaling the stdev
        self.std_scaling = 4.0
        self.classifier_regr_std = [8.0, 8.0, 4.0, 4.0]

        # overlaps for RPN
        self.rpn_min_overlap = 0.3
        self.rpn_max_overlap = 0.7

        # overlaps for classifier ROIs
        self.classifier_min_overlap = 0.1
        self.classifier_max_overlap = 0.5

        # placeholder for the class mapping, automatically generated by the parser
        self.class_mapping = None

        self.model_path = None


# #### Parser the data from annotation file

def get_data(input_path):
    """Parse the data from annotation file

    Args:
        input_path: annotation file path

    Returns:
        all_data: list(filepath, width, height, list(bboxes))
        classes_count: dict{key:class_name, value:count_num}
            e.g. {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745}
        class_mapping: dict{key:class_name, value: idx}
            e.g. {'Car': 0, 'Mobile phone': 1, 'Person': 2}
```

```
    """
    found_bg = False
    all_imgs = {}

    classes_count = {}

    class_mapping = {}

    visualise = True

    i = 1

    with open(input_path, 'r') as f:

        print('Parsing annotation files')

        for line in f:

            # Print process
            sys.stdout.write('\r' + 'idx=' + str(i))
            i += 1

            line_split = line.strip().split(',')
            if line_split[0]=='filename':
              i = 1
              continue
            # Make sure the info saved in annotation file matching the format
(path_filename, x1, y1, x2, y2, class_name)
            # Note:
            #    One path_filename might has several classes (class_name)
            #    x1, y1, x2, y2 are the pixel value of the origial image, not the ratio
value
            #    (x1, y1) top left coordinates; (x2, y2) bottom right coordinates
            #   x1,y1-------------------
            #    |                                 |
            #    |                                 |
            #    |                                 |
            #    |                                 |
            #    ---------------------x2,y2

            (filename, x1, y1, x2, y2, class_name) = line_split

            if class_name not in classes_count:
                classes_count[class_name] = 1
            else:
                classes_count[class_name] += 1

            if class_name not in class_mapping:
                if class_name == 'bg' and found_bg == False:
                    print(
                        'Found class name with special name bg. Will be treated as a
background region (this is usually for hard negative mining).')
                    found_bg = True
                class_mapping[class_name] = len(class_mapping)

            if filename not in all_imgs:
                all_imgs[filename] = {}
                print('\n'+os.getcwd())
                print(filename)
                img = cv2.imread(filename)
                (rows, cols) = img.shape[:2]
                all_imgs[filename]['filepath'] = filename
                all_imgs[filename]['width'] = cols
                all_imgs[filename]['height'] = rows
                all_imgs[filename]['bboxes'] = []
```

```
            # if np.random.randint(0,6) > 0:
            #    all_imgs[filename]['imageset'] = 'trainval'
            # else:
            #    all_imgs[filename]['imageset'] = 'test'

            all_imgs[filename]['bboxes'].append(
                {'class': class_name, 'x1': int(x1), 'x2': int(x2), 'y1': int(y1),
'y2': int(y2)})

        all_data = []
        for key in all_imgs:
            all_data.append(all_imgs[key])

        # make sure the bg class is last in the list
        if found_bg:
            if class_mapping['bg'] != len(class_mapping) - 1:
                key_to_switch = [key for key in class_mapping.keys() if
class_mapping[key] == len(class_mapping) - 1][0]
                val_to_switch = class_mapping['bg']
                class_mapping['bg'] = len(class_mapping) - 1
                class_mapping[key_to_switch] = val_to_switch

        return all_data, classes_count, class_mapping


# #### Define ROI Pooling Convolutional Layer




class RoiPoolingConv(Layer):
    '''ROI pooling layer for 2D inputs.
    See Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,
    K. He, X. Zhang, S. Ren, J. Sun
    # Arguments
        pool_size: int
            Size of pooling region to use. pool_size = 7 will result in a 7x7 region.
        num_rois: number of regions of interest to be used
    # Input shape
        list of two 4D tensors [X_img,X_roi] with shape:
        X_img:
        `(1, rows, cols, channels)`
        X_roi:
        `(1,num_rois,4)` list of rois, with ordering (x,y,w,h)
    # Output shape
        3D tensor with shape:
        `(1, num_rois, channels, pool_size, pool_size)`
    '''
    def __init__(self, pool_size, num_rois, **kwargs):

        self.dim_ordering = K.image_data_format()
        self.pool_size = pool_size
        self.num_rois = num_rois

        super(RoiPoolingConv, self).__init__(**kwargs)

    def build(self, input_shape):
        self.nb_channels = input_shape[0][3]

    def compute_output_shape(self, input_shape):
        return None, self.num_rois, self.pool_size, self.pool_size, self.nb_channels

    def call(self, x, mask=None):

        assert(len(x) == 2)
```

```
        # x[0] is image with shape (rows, cols, channels)
        img = x[0]

        # x[1] is roi with shape (num_rois,4) with ordering (x,y,w,h)
        rois = x[1]

        input_shape = K.shape(img)

        outputs = []

        for roi_idx in range(self.num_rois):

            x = rois[0, roi_idx, 0]
            y = rois[0, roi_idx, 1]
            w = rois[0, roi_idx, 2]
            h = rois[0, roi_idx, 3]

            x = K.cast(x, 'int32')
            y = K.cast(y, 'int32')
            w = K.cast(w, 'int32')
            h = K.cast(h, 'int32')

            # Resized roi of the image to pooling size (7x7)
            rs = tf.image.resize_images(img[:, y:y+h, x:x+w, :], (self.pool_size,
self.pool_size))
            outputs.append(rs)


        final_output = K.concatenate(outputs, axis=0)

        # Reshape to (1, num_rois, pool_size, pool_size, nb_channels)
        # Might be (1, 4, 7, 7, 3)
        final_output = K.reshape(final_output, (1, self.num_rois, self.pool_size,
self.pool_size, self.nb_channels))

        # permute_dimensions is similar to transpose
        final_output = K.permute_dimensions(final_output, (0, 1, 2, 3, 4))

        return final_output


    def get_config(self):
        config = {'pool_size': self.pool_size,
                  'num_rois': self.num_rois}
        base_config = super(RoiPoolingConv, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))


# #### Vgg-16 model




def get_img_output_length(width, height):
    def get_output_length(input_length):
        return input_length//16

    return get_output_length(width), get_output_length(height)

def nn_base(input_tensor=None, trainable=False):


    input_shape = (None, None, 3)
```

```
    if input_tensor is None:
        img_input = Input(shape=input_shape)
    else:
        if not K.is_keras_tensor(input_tensor):
            img_input = Input(tensor=input_tensor, shape=input_shape)
        else:
            img_input = input_tensor

    bn_axis = 3

    # Block 1
    x = Conv2D(64, (3, 3), activation='relu', padding='same',
name='block1_conv1')(img_input)
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

    # Block 2
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

    # Block 3
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    #x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)
    print(x.shape)
    # x = Flatten()(x)
    # x = Dense(units=4096,activation="relu")(x)
    # x = Dense(units=4096,activation="relu")(x)
    # x = Dense(units=1, activation="softmax")(x)

    return x


# ####  RPN layer




def rpn_layer(base_layers, num_anchors):
    """Create a rpn layer
        Step1: Pass through the feature map from base layer to a 3x3 512 channels
convolutional layer
                Keep the padding 'same' to preserve the feature map's size
        Step2: Pass the step1 to two (1,1) convolutional layer to replace the fully
connected layer
                classification layer: num_anchors (9 in here) channels for 0, 1
sigmoid activation output
                regression layer: num_anchors*4 (36 in here) channels for computing
the regression of bboxes with linear activation
    Args:
        base_layers: vgg in here
```

```
        num_anchors: 9 in here

    Returns:
        [x_class, x_regr, base_layers]
        x_class: classification for whether it's an object
        x_regr: bboxes regression
        base_layers: vgg in here
    """
    x = Conv2D(512, (3, 3), padding='same', activation='relu',
kernel_initializer='normal', name='rpn_conv1')(base_layers)

    x_class = Conv2D(num_anchors, (1, 1), activation='sigmoid',
kernel_initializer='uniform', name='rpn_out_class')(x)
    x_regr = Conv2D(num_anchors * 4, (1, 1), activation='linear',
kernel_initializer='zero', name='rpn_out_regress')(x)

    return [x_class, x_regr, base_layers]


# ####  Classifier layer




def classifier_layer(base_layers, input_rois, num_rois, nb_classes = 4):
    """Create a classifier layer

    Args:
        base_layers: vgg
        input_rois: `(1,num_rois,4)` list of rois, with ordering (x,y,w,h)
        num_rois: number of rois to be processed in one time (4 in here)

    Returns:
        list(out_class, out_regr)
        out_class: classifier layer output
        out_regr: regression layer output
    """

    input_shape = (num_rois,7,7,512)

    pooling_regions = 7

    # out_roi_pool.shape = (1, num_rois, channels, pool_size, pool_size)
    # num_rois (4) 7x7 roi pooling
    out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers,
input_rois])

    # Flatten the convlutional layer and connected to 2 FC and 2 dropout
    out = TimeDistributed(Flatten(name='flatten'))(out_roi_pool)
    out = TimeDistributed(Dense(4096, activation='relu', name='fc1'))(out)
    out = TimeDistributed(Dropout(0.5))(out)
    out = TimeDistributed(Dense(4096, activation='relu', name='fc2'))(out)
    out = TimeDistributed(Dropout(0.5))(out)
    # out = TimeDistributed(Dense(4096, activation='relu', name='fc3'))(out)
    # out = TimeDistributed(Dropout(0.5))(out)

    # There are two output layer
    # out_class: softmax acivation function for classify the class name of the object
    # out_regr: linear activation function for bboxes coordinates regression
    out_class = TimeDistributed(Dense(nb_classes, activation='softmax',
kernel_initializer='zero'), name='dense_class_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out_regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
kernel_initializer='zero'), name='dense_regress_{}'.format(nb_classes))(out)
```

```
    return [out_class, out_regr]


# #### Calculate IoU (Intersection of Union)




def union(au, bu, area_intersection):
	area_a = (au[2] - au[0]) * (au[3] - au[1])
	area_b = (bu[2] - bu[0]) * (bu[3] - bu[1])
	area_union = area_a + area_b - area_intersection
	return area_union


def intersection(ai, bi):
	x = max(ai[0], bi[0])
	y = max(ai[1], bi[1])
	w = min(ai[2], bi[2]) - x
	h = min(ai[3], bi[3]) - y
	if w < 0 or h < 0:
		return 0
	return w*h


def iou(a, b):
	# a and b should be (x1,y1,x2,y2)

	if a[0] >= a[2] or a[1] >= a[3] or b[0] >= b[2] or b[1] >= b[3]:
		return 0.0

	area_i = intersection(a, b)
	area_u = union(a, b, area_i)

	return float(area_i) / float(area_u + 1e-6)


# #### Calculate the rpn for all anchors of all images




def calc_rpn(C, img_data, width, height, resized_width, resized_height,
img_length_calc_function):
	"""(Important part!) Calculate the rpn for all anchors
		If feature map has shape 38x50=1900, there arae 1900x9=17100 potential
anchors

	Args:
		C: config
		img_data: augmented image data
		width: original image width (e.g. 600)
		height: original image height (e.g. 800)
		resized_width: resized image width according to C.im_size (e.g. 300)
		resized_height: resized image height according to C.im_size (e.g. 400)
		img_length_calc_function: function to calculate final layer's feature map
(of base model) size according to input image size

	Returns:
		y_rpn_cls: list(num_bboxes, y_is_box_valid + y_rpn_overlap)
			y_is_box_valid: 0 or 1 (0 means the box is invalid, 1 means the box is
valid)
			y_rpn_overlap: 0 or 1 (0 means the box is not an object, 1 means the
box is an object)
		y_rpn_regr: list(num_bboxes, 4*y_rpn_overlap + y_rpn_regr)
```

```
            y_rpn_regr: x1,y1,x2,y2 bunding boxes coordinates
    """
    downscale = float(C.rpn_stride)
    anchor_sizes = C.anchor_box_scales   # 128, 256, 512
    anchor_ratios = C.anchor_box_ratios  # 1:1, 1:2*sqrt(2), 2*sqrt(2):1
    num_anchors = len(anchor_sizes) * len(anchor_ratios) # 3x3=9

    # calculate the output map size based on the network architecture
    (output_width, output_height) = img_length_calc_function(resized_width,
resized_height)

    n_anchratios = len(anchor_ratios)    # 3

    # initialise empty output objectives
    y_rpn_overlap = np.zeros((output_height, output_width, num_anchors))
    y_is_box_valid = np.zeros((output_height, output_width, num_anchors))
    y_rpn_regr = np.zeros((output_height, output_width, num_anchors * 4))

    num_bboxes = len(img_data['bboxes'])

    num_anchors_for_bbox = np.zeros(num_bboxes).astype(int)
    best_anchor_for_bbox = -1*np.ones((num_bboxes, 4)).astype(int)
    best_iou_for_bbox = np.zeros(num_bboxes).astype(np.float32)
    best_x_for_bbox = np.zeros((num_bboxes, 4)).astype(int)
    best_dx_for_bbox = np.zeros((num_bboxes, 4)).astype(np.float32)

    # get the GT box coordinates, and resize to account for image resizing
    gta = np.zeros((num_bboxes, 4))
    for bbox_num, bbox in enumerate(img_data['bboxes']):
        # get the GT box coordinates, and resize to account for image resizing
        gta[bbox_num, 0] = bbox['x1'] * (resized_width / float(width))
        gta[bbox_num, 1] = bbox['x2'] * (resized_width / float(width))
        gta[bbox_num, 2] = bbox['y1'] * (resized_height / float(height))
        gta[bbox_num, 3] = bbox['y2'] * (resized_height / float(height))

    # rpn ground truth

    for anchor_size_idx in range(len(anchor_sizes)):
        for anchor_ratio_idx in range(n_anchratios):
            anchor_x = anchor_sizes[anchor_size_idx] *
anchor_ratios[anchor_ratio_idx][0]
            anchor_y = anchor_sizes[anchor_size_idx] *
anchor_ratios[anchor_ratio_idx][1]

            for ix in range(output_width):
                # x-coordinates of the current anchor box
                x1_anc = downscale * (ix + 0.5) - anchor_x / 2
                x2_anc = downscale * (ix + 0.5) + anchor_x / 2

                # ignore boxes that go across image boundaries

                if x1_anc < 0 or x2_anc > resized_width:
                    continue

                for jy in range(output_height):

                    # y-coordinates of the current anchor box
                    y1_anc = downscale * (jy + 0.5) - anchor_y / 2
                    y2_anc = downscale * (jy + 0.5) + anchor_y / 2

                    # ignore boxes that go across image boundaries
                    if y1_anc < 0 or y2_anc > resized_height:
                        continue

                    # bbox_type indicates whether an anchor should be a target
```

```
                            # Initialize with 'negative'
                            bbox_type = 'neg'

                            # this is the best IOU for the (x,y) coord and the current
anchor
                            # note that this is different from the best IOU for a GT
bbox
                            best_iou_for_loc = 0.0

                            for bbox_num in range(num_bboxes):

                                  # get IOU of the current GT box and the current
anchor box
                                  curr_iou = iou([gta[bbox_num, 0], gta[bbox_num, 2],
gta[bbox_num, 1], gta[bbox_num, 3]], [x1_anc, y1_anc, x2_anc, y2_anc])
                                  # calculate the regression targets if they will be
needed
                                  if curr_iou > best_iou_for_bbox[bbox_num] or curr_iou
> C.rpn_max_overlap:
                                        cx = (gta[bbox_num, 0] + gta[bbox_num, 1]) / 2.0
                                        cy = (gta[bbox_num, 2] + gta[bbox_num, 3]) / 2.0
                                        cxa = (x1_anc + x2_anc)/2.0
                                        cya = (y1_anc + y2_anc)/2.0

                                        # x,y are the center point of ground-truth bbox
                                        # xa,ya are the center point of anchor bbox
(xa=downscale * (ix + 0.5); ya=downscale * (iy+0.5))
                                        # w,h are the width and height of ground-truth
bbox
                                        # wa,ha are the width and height of anchor bboxe
                                        # tx = (x - xa) / wa
                                        # ty = (y - ya) / ha
                                        # tw = log(w / wa)
                                        # th = log(h / ha)
                                        tx = (cx - cxa) / (x2_anc - x1_anc)
                                        ty = (cy - cya) / (y2_anc - y1_anc)
                                        tw = np.log((gta[bbox_num, 1] - gta[bbox_num,
0]) / (x2_anc - x1_anc))
                                        th = np.log((gta[bbox_num, 3] - gta[bbox_num,
2]) / (y2_anc - y1_anc))

                                  if img_data['bboxes'][bbox_num]['class'] != 'bg':

                                        # all GT boxes should be mapped to an anchor
box, so we keep track of which anchor box was best
                                        if curr_iou > best_iou_for_bbox[bbox_num]:
                                              best_anchor_for_bbox[bbox_num] = [jy, ix,
anchor_ratio_idx, anchor_size_idx]
                                              best_iou_for_bbox[bbox_num] = curr_iou
                                              best_x_for_bbox[bbox_num,:] = [x1_anc,
x2_anc, y1_anc, y2_anc]
                                              best_dx_for_bbox[bbox_num,:] = [tx, ty,
tw, th]

                                        # we set the anchor to positive if the IOU is
>0.7 (it does not matter if there was another better box, it just indicates overlap)
                                        if curr_iou > C.rpn_max_overlap:
                                              bbox_type = 'pos'
                                              num_anchors_for_bbox[bbox_num] += 1
                                              # we update the regression layer target if
this IOU is the best for the current (x,y) and anchor position
                                              if curr_iou > best_iou_for_loc:
                                                    best_iou_for_loc = curr_iou
                                                    best_regr = (tx, ty, tw, th)
```

```
                                    # if the IOU is >0.3 and <0.7, it is ambiguous
and no included in the objective
                                    if C.rpn_min_overlap < curr_iou <
C.rpn_max_overlap:
                                         # gray zone between neg and pos
                                         if bbox_type != 'pos':
                                              bbox_type = 'neutral'

                          # turn on or off outputs depending on IOUs
                          if bbox_type == 'neg':
                               y_is_box_valid[jy, ix, anchor_ratio_idx +
n_anchratios * anchor_size_idx] = 1
                               y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios
* anchor_size_idx] = 0
                          elif bbox_type == 'neutral':
                               y_is_box_valid[jy, ix, anchor_ratio_idx +
n_anchratios * anchor_size_idx] = 0
                               y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios
* anchor_size_idx] = 0
                          elif bbox_type == 'pos':
                               y_is_box_valid[jy, ix, anchor_ratio_idx +
n_anchratios * anchor_size_idx] = 1
                               y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios
* anchor_size_idx] = 1
                               start = 4 * (anchor_ratio_idx + n_anchratios *
anchor_size_idx)
                               y_rpn_regr[jy, ix, start:start+4] = best_regr

     # we ensure that every bbox has at least one positive RPN region

     for idx in range(num_anchors_for_bbox.shape[0]):
          if num_anchors_for_bbox[idx] == 0:
               # no box with an IOU greater than zero ...
               if best_anchor_for_bbox[idx, 0] == -1:
                    continue
               y_is_box_valid[
                    best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
best_anchor_for_bbox[idx,2] + n_anchratios *
                    best_anchor_for_bbox[idx,3]] = 1
               y_rpn_overlap[
                    best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
best_anchor_for_bbox[idx,2] + n_anchratios *
                    best_anchor_for_bbox[idx,3]] = 1
               start = 4 * (best_anchor_for_bbox[idx,2] + n_anchratios *
best_anchor_for_bbox[idx,3])
               y_rpn_regr[
                    best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
start:start+4] = best_dx_for_bbox[idx, :]

     y_rpn_overlap = np.transpose(y_rpn_overlap, (2, 0, 1))
     y_rpn_overlap = np.expand_dims(y_rpn_overlap, axis=0)

     y_is_box_valid = np.transpose(y_is_box_valid, (2, 0, 1))
     y_is_box_valid = np.expand_dims(y_is_box_valid, axis=0)

     y_rpn_regr = np.transpose(y_rpn_regr, (2, 0, 1))
     y_rpn_regr = np.expand_dims(y_rpn_regr, axis=0)

     pos_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 1,
y_is_box_valid[0, :, :, :] == 1))
     neg_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 0,
y_is_box_valid[0, :, :, :] == 1))

     num_pos = len(pos_locs[0])
```

```
	# one issue is that the RPN has many more negative than positive regions, so we
turn off some of the negative
	# regions. We also limit it to 256 regions.
	num_regions = 256

	if len(pos_locs[0]) > num_regions/2:
		val_locs = random.sample(range(len(pos_locs[0])), len(pos_locs[0]) -
num_regions/2)
		y_is_box_valid[0, pos_locs[0][val_locs], pos_locs[1][val_locs],
pos_locs[2][val_locs]] = 0
		num_pos = num_regions/2

	if len(neg_locs[0]) + num_pos > num_regions:
		val_locs = random.sample(range(len(neg_locs[0])), len(neg_locs[0]) -
num_pos)
		y_is_box_valid[0, neg_locs[0][val_locs], neg_locs[1][val_locs],
neg_locs[2][val_locs]] = 0

	y_rpn_cls = np.concatenate([y_is_box_valid, y_rpn_overlap], axis=1)
	y_rpn_regr = np.concatenate([np.repeat(y_rpn_overlap, 4, axis=1), y_rpn_regr],
axis=1)

	return np.copy(y_rpn_cls), np.copy(y_rpn_regr), num_pos


# #### Get new image size and augment the image




def get_new_img_size(width, height, img_min_side=1024):
	if width <= height:
		f = float(img_min_side) / width
		resized_height = int(f * height)
		resized_width = img_min_side
	else:
		f = float(img_min_side) / height
		resized_width = int(f * width)
		resized_height = img_min_side

	return resized_width, resized_height

def augment(img_data, config, augment=True):
	assert 'filepath' in img_data
	assert 'bboxes' in img_data
	assert 'width' in img_data
	assert 'height' in img_data

	img_data_aug = copy.deepcopy(img_data)

	img = cv2.imread(img_data_aug['filepath'])

	if augment:
		rows, cols = img.shape[:2]

		if config.use_horizontal_flips and np.random.randint(0, 2) == 0:
			img = cv2.flip(img, 1)
			for bbox in img_data_aug['bboxes']:
				x1 = bbox['x1']
				x2 = bbox['x2']
				bbox['x2'] = cols - x1
				bbox['x1'] = cols - x2

		if config.use_vertical_flips and np.random.randint(0, 2) == 0:
			img = cv2.flip(img, 0)
```

```
                for bbox in img_data_aug['bboxes']:
                        y1 = bbox['y1']
                        y2 = bbox['y2']
                        bbox['y2'] = rows - y1
                        bbox['y1'] = rows - y2

        if config.rot_90:
                angle = np.random.choice([0,90,180,270],1)[0]
                if angle == 270:
                        img = np.transpose(img, (1,0,2))
                        img = cv2.flip(img, 0)
                elif angle == 180:
                        img = cv2.flip(img, -1)
                elif angle == 90:
                        img = np.transpose(img, (1,0,2))
                        img = cv2.flip(img, 1)
                elif angle == 0:
                        pass

                for bbox in img_data_aug['bboxes']:
                        x1 = bbox['x1']
                        x2 = bbox['x2']
                        y1 = bbox['y1']
                        y2 = bbox['y2']
                        if angle == 270:
                                bbox['x1'] = y1
                                bbox['x2'] = y2
                                bbox['y1'] = cols - x2
                                bbox['y2'] = cols - x1
                        elif angle == 180:
                                bbox['x2'] = cols - x1
                                bbox['x1'] = cols - x2
                                bbox['y2'] = rows - y1
                                bbox['y1'] = rows - y2
                        elif angle == 90:
                                bbox['x1'] = rows - y2
                                bbox['x2'] = rows - y1
                                bbox['y1'] = x1
                                bbox['y2'] = x2
                        elif angle == 0:
                                pass

        img_data_aug['width'] = img.shape[1]
        img_data_aug['height'] = img.shape[0]
        return img_data_aug, img


# #### Generate the ground_truth anchors




def get_anchor_gt(all_img_data, C, img_length_calc_function, mode='train'):
        """ Yield the ground-truth anchors as Y (labels)

        Args:
                all_img_data: list(filepath, width, height, list(bboxes))
                C: config
                img_length_calc_function: function to calculate final layer's feature map
(of base model) size according to input image size
                mode: 'train' or 'test'; 'train' mode need augmentation

        Returns:
                x_img: image data after resized and scaling (smallest size = 300px)
                Y: [y_rpn_cls, y_rpn_regr]
```

```
            img_data_aug: augmented image data (original image with augmentation)
            debug_img: show image for debug
            num_pos: show number of positive anchors for debug
      """
      while True:

            for img_data in all_img_data:
                  try:

                        # read in image, and optionally add augmentation

                        if mode == 'train':
                              img_data_aug, x_img = augment(img_data, C, augment=True)
                        else:
                              img_data_aug, x_img = augment(img_data, C, augment=False)

                        (width, height) = (img_data_aug['width'],
img_data_aug['height'])
                        (rows, cols, _) = x_img.shape

                        assert cols == width
                        assert rows == height

                        # get image dimensions for resizing
                        (resized_width, resized_height) = get_new_img_size(width,
height, C.im_size)

                        # resize the image so that smalles side is length = 300px
                        #x_img = cv2.resize(x_img, (resized_width, resized_height),
interpolation=cv2.INTER_CUBIC)
                        debug_img = x_img.copy()

                        try:
                              y_rpn_cls, y_rpn_regr, num_pos = calc_rpn(C, img_data_aug,
width, height, resized_width, resized_height, img_length_calc_function)
                        except:
                              continue

                        # Zero-center by mean pixel, and preprocess image

                        x_img = x_img[:,:, (2, 1, 0)]  # BGR -> RGB
                        x_img = x_img.astype(np.float32)
                        x_img[:, :, 0] -= C.img_channel_mean[0]
                        x_img[:, :, 1] -= C.img_channel_mean[1]
                        x_img[:, :, 2] -= C.img_channel_mean[2]
                        x_img /= C.img_scaling_factor

                        x_img = np.transpose(x_img, (2, 0, 1))
                        x_img = np.expand_dims(x_img, axis=0)

                        y_rpn_regr[:, y_rpn_regr.shape[1]//2:, :, :] *= C.std_scaling

                        x_img = np.transpose(x_img, (0, 2, 3, 1))
                        y_rpn_cls = np.transpose(y_rpn_cls, (0, 2, 3, 1))
                        y_rpn_regr = np.transpose(y_rpn_regr, (0, 2, 3, 1))

                        yield np.copy(x_img), [np.copy(y_rpn_cls), np.copy(y_rpn_regr)],
img_data_aug, debug_img, num_pos

                  except Exception as e:
                        print(e)
                        continue


# #### Define loss functions for all four outputs
```

```
lambda_rpn_regr = 1.0
lambda_rpn_class = 1.0

lambda_cls_regr = 1.0
lambda_cls_class = 1.0

epsilon = 1e-5




def rpn_loss_regr(num_anchors):
    """Loss function for rpn regression
    Args:
        num_anchors: number of anchors (9 in here)
    Returns:
        Smooth L1 loss function
                           0.5*x*x (if x_abs < 1)
                           x_abx - 0.5 (otherwise)
    """
    def rpn_loss_regr_fixed_num(y_true, y_pred):

        # x is the difference between true value and predicted vaue
        x = y_true[:, :, :, 4 * num_anchors:] - y_pred

        # absolute value of x
        x_abs = K.abs(x)

        # If x_abs <= 1.0, x_bool = 1
        x_bool = K.cast(K.less_equal(x_abs, 1.0), tf.float32)

        return lambda_rpn_regr * K.sum(
            y_true[:, :, :, :4 * num_anchors] * (x_bool * (0.5 * x * x) + (1 - x_bool)
* (x_abs - 0.5))) / K.sum(epsilon + y_true[:, :, :, :4 * num_anchors])

    return rpn_loss_regr_fixed_num


def rpn_loss_cls(num_anchors):
    """Loss function for rpn classification
    Args:
        num_anchors: number of anchors (9 in here)
        y_true[:, :, :, :9]: [0,1,0,0,0,0,0,1,0] means only the second and the eighth
box is valid which contains pos or neg anchor => isValid
        y_true[:, :, :, 9:]: [0,1,0,0,0,0,0,0,0] means the second box is pos and
eighth box is negative
    Returns:
        lambda * sum((binary_crossentropy(isValid*y_pred,y_true))) / N
    """
    def rpn_loss_cls_fixed_num(y_true, y_pred):

            return lambda_rpn_class * K.sum(y_true[:, :, :, :num_anchors] *
K.binary_crossentropy(y_pred[:, :, :, :], y_true[:, :, :, num_anchors:])) /
K.sum(epsilon + y_true[:, :, :, :num_anchors])

    return rpn_loss_cls_fixed_num


def class_loss_regr(num_classes):
    """Loss function for rpn regression
```

```
    Args:
        num_anchors: number of anchors (9 in here)
    Returns:
        Smooth L1 loss function
                           0.5*x*x (if x_abs < 1)
                           x_abx - 0.5 (otherwise)
    """
    def class_loss_regr_fixed_num(y_true, y_pred):
        x = y_true[:, :, 4*num_classes:] - y_pred
        x_abs = K.abs(x)
        x_bool = K.cast(K.less_equal(x_abs, 1.0), 'float32')
        return lambda_cls_regr * K.sum(y_true[:, :, :4*num_classes] * (x_bool * (0.5 *
x * x) + (1 - x_bool) * (x_abs - 0.5))) / K.sum(epsilon + y_true[:, :,
:4*num_classes])
    return class_loss_regr_fixed_num


def class_loss_cls(y_true, y_pred):
    return lambda_cls_class * K.mean(categorical_crossentropy(y_true[0, :, :],
y_pred[0, :, :]))




def non_max_suppression_fast(boxes, probs, overlap_thresh=0.9, max_boxes=300):
    # code used from here: http://www.pyimagesearch.com/2015/02/16/faster-non-maximum-
suppression-python/
    # if there are no boxes, return an empty list

    # Process explanation:
    #   Step 1: Sort the probs list
    #   Step 2: Find the larget prob 'Last' in the list and save it to the pick list
    #   Step 3: Calculate the IoU with 'Last' box and other boxes in the list. If the
IoU is larger than overlap_threshold, delete the box from list
    #   Step 4: Repeat step 2 and step 3 until there is no item in the probs list
    if len(boxes) == 0:
        return []

    # grab the coordinates of the bounding boxes
    x1 = boxes[:, 0]
    y1 = boxes[:, 1]
    x2 = boxes[:, 2]
    y2 = boxes[:, 3]

    np.testing.assert_array_less(x1, x2)
    np.testing.assert_array_less(y1, y2)

    # if the bounding boxes integers, convert them to floats --
    # this is important since we'll be doing a bunch of divisions
    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")

    # initialize the list of picked indexes
    pick = []

    # calculate the areas
    area = (x2 - x1) * (y2 - y1)

    # sort the bounding boxes
    idxs = np.argsort(probs)

    # keep looping while some indexes still remain in the indexes
    # list
    while len(idxs) > 0:
```

```
        # grab the last index in the indexes list and add the
        # index value to the list of picked indexes
        last = len(idxs) - 1
        i = idxs[last]
        pick.append(i)

        # find the intersection

        xx1_int = np.maximum(x1[i], x1[idxs[:last]])
        yy1_int = np.maximum(y1[i], y1[idxs[:last]])
        xx2_int = np.minimum(x2[i], x2[idxs[:last]])
        yy2_int = np.minimum(y2[i], y2[idxs[:last]])

        ww_int = np.maximum(0, xx2_int - xx1_int)
        hh_int = np.maximum(0, yy2_int - yy1_int)

        area_int = ww_int * hh_int

        # find the union
        area_union = area[i] + area[idxs[:last]] - area_int

        # compute the ratio of overlap
        overlap = area_int/(area_union + 1e-6)

        # delete all indexes from the index list that have
        idxs = np.delete(idxs, np.concatenate(([last],
            np.where(overlap > overlap_thresh)[0])))

        if len(pick) >= max_boxes:
            break

    # return only the bounding boxes that were picked using the integer data type
    boxes = boxes[pick].astype("int")
    probs = probs[pick]
    return boxes, probs

def apply_regr_np(X, T):
    """Apply regression layer to all anchors in one feature map

    Args:
        X: shape=(4, 18, 25) the current anchor type for all points in the feature map
        T: regression layer shape=(4, 18, 25)

    Returns:
        X: regressed position and size for current anchor
    """
    try:
        x = X[0, :, :]
        y = X[1, :, :]
        w = X[2, :, :]
        h = X[3, :, :]

        tx = T[0, :, :]
        ty = T[1, :, :]
        tw = T[2, :, :]
        th = T[3, :, :]

        cx = x + w/2.
        cy = y + h/2.
        cx1 = tx * w + cx
        cy1 = ty * h + cy

        w1 = np.exp(tw.astype(np.float64)) * w
        h1 = np.exp(th.astype(np.float64)) * h
        x1 = cx1 - w1/2.
```

```
        y1 = cy1 - h1/2.

        x1 = np.round(x1)
        y1 = np.round(y1)
        w1 = np.round(w1)
        h1 = np.round(h1)
        return np.stack([x1, y1, w1, h1])
    except Exception as e:
        print(e)
        return X

def apply_regr(x, y, w, h, tx, ty, tw, th):
    # Apply regression to x, y, w and h
    try:
        cx = x + w/2.
        cy = y + h/2.
        cx1 = tx * w + cx
        cy1 = ty * h + cy
        w1 = math.exp(tw) * w
        h1 = math.exp(th) * h
        x1 = cx1 - w1/2.
        y1 = cy1 - h1/2.
        x1 = int(round(x1))
        y1 = int(round(y1))
        w1 = int(round(w1))
        h1 = int(round(h1))

        return x1, y1, w1, h1

    except ValueError:
        return x, y, w, h
    except OverflowError:
        return x, y, w, h
    except Exception as e:
        print(e)
        return x, y, w, h

def calc_iou(R, img_data, C, class_mapping):
    """Converts from (x1,y1,x2,y2) to (x,y,w,h) format

    Args:
        R: bboxes, probs
    """
    bboxes = img_data['bboxes']
    (width, height) = (img_data['width'], img_data['height'])
    # get image dimensions for resizing
    (resized_width, resized_height) = get_new_img_size(width, height, C.im_size)

    gta = np.zeros((len(bboxes), 4))

    for bbox_num, bbox in enumerate(bboxes):
        # get the GT box coordinates, and resize to account for image resizing
        # gta[bbox_num, 0] = (40 * (600 / 800)) / 16 = int(round(1.875)) = 2 (x in
feature map)
        gta[bbox_num, 0] = int(round(bbox['x1'] * (resized_width /
float(width))/C.rpn_stride))
        gta[bbox_num, 1] = int(round(bbox['x2'] * (resized_width /
float(width))/C.rpn_stride))
        gta[bbox_num, 2] = int(round(bbox['y1'] * (resized_height /
float(height))/C.rpn_stride))
        gta[bbox_num, 3] = int(round(bbox['y2'] * (resized_height /
float(height))/C.rpn_stride))

    x_roi = []
    y_class_num = []
```

```
    y_class_regr_coords = []
    y_class_regr_label = []
    IoUs = [] # for debugging only

    # R.shape[0]: number of bboxes (=300 from non_max_suppression)
    for ix in range(R.shape[0]):
        (x1, y1, x2, y2) = R[ix, :]
        x1 = int(round(x1))
        y1 = int(round(y1))
        x2 = int(round(x2))
        y2 = int(round(y2))

        best_iou = 0.0
        best_bbox = -1
        # Iterate through all the ground-truth bboxes to calculate the iou
        for bbox_num in range(len(bboxes)):
            curr_iou = iou([gta[bbox_num, 0], gta[bbox_num, 2], gta[bbox_num, 1],
gta[bbox_num, 3]], [x1, y1, x2, y2])

            # Find out the corresponding ground-truth bbox_num with larget iou
            if curr_iou > best_iou:
                best_iou = curr_iou
                best_bbox = bbox_num

        if best_iou < C.classifier_min_overlap:
                continue
        else:
            w = x2 - x1
            h = y2 - y1
            x_roi.append([x1, y1, w, h])
            IoUs.append(best_iou)

            if C.classifier_min_overlap <= best_iou < C.classifier_max_overlap:
                # hard negative example
                cls_name = 'bg'
            elif C.classifier_max_overlap <= best_iou:
                cls_name = bboxes[best_bbox]['class']
                cxg = (gta[best_bbox, 0] + gta[best_bbox, 1]) / 2.0
                cyg = (gta[best_bbox, 2] + gta[best_bbox, 3]) / 2.0

                cx = x1 + w / 2.0
                cy = y1 + h / 2.0

                tx = (cxg - cx) / float(w)
                ty = (cyg - cy) / float(h)
                tw = np.log((gta[best_bbox, 1] - gta[best_bbox, 0]) / float(w))
                th = np.log((gta[best_bbox, 3] - gta[best_bbox, 2]) / float(h))
            else:
                print('roi = {}'.format(best_iou))
                raise RuntimeError

        class_num = class_mapping[cls_name]
        class_label = len(class_mapping) * [0]
        class_label[class_num] = 1
        y_class_num.append(copy.deepcopy(class_label))
        coords = [0] * 4 * (len(class_mapping) - 1)
        labels = [0] * 4 * (len(class_mapping) - 1)
        if cls_name != 'bg':
            label_pos = 4 * class_num
            sx, sy, sw, sh = C.classifier_regr_std
            coords[label_pos:4+label_pos] = [sx*tx, sy*ty, sw*tw, sh*th]
            labels[label_pos:4+label_pos] = [1, 1, 1, 1]
            y_class_regr_coords.append(copy.deepcopy(coords))
            y_class_regr_label.append(copy.deepcopy(labels))
        else:
```

```
            y_class_regr_coords.append(copy.deepcopy(coords))
            y_class_regr_label.append(copy.deepcopy(labels))

    if len(x_roi) == 0:
        return None, None, None, None

    # bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300
non_max_suppression bboxes
    X = np.array(x_roi)
    # one hot code for bboxes from above => x_roi (X)
    Y1 = np.array(y_class_num)
    # corresponding labels and corresponding gt bboxes
    Y2 =
np.concatenate([np.array(y_class_regr_label),np.array(y_class_regr_coords)],axis=1)

    return np.expand_dims(X, axis=0), np.expand_dims(Y1, axis=0), np.expand_dims(Y2,
axis=0), IoUs




def rpn_to_roi(rpn_layer, regr_layer, C, dim_ordering, use_regr=True,
max_boxes=300,overlap_thresh=0.9):
      """Convert rpn layer to roi bboxes

      Args: (num_anchors = 9)
            rpn_layer: output layer for rpn classification
                  shape (1, feature_map.height, feature_map.width, num_anchors)
                  Might be (1, 18, 25, 18) if resized image is 400 width and 300
            regr_layer: output layer for rpn regression
                  shape (1, feature_map.height, feature_map.width, num_anchors)
                  Might be (1, 18, 25, 72) if resized image is 400 width and 300
            C: config
            use_regr: Wether to use bboxes regression in rpn
            max_boxes: max bboxes number for non-max-suppression (NMS)
            overlap_thresh: If iou in NMS is larger than this threshold, drop the box

      Returns:
            result: boxes from non-max-suppression (shape=(300, 4))
                  boxes: coordinates for bboxes (on the feature map)
      """
      regr_layer = regr_layer / C.std_scaling

      anchor_sizes = C.anchor_box_scales   # (3 in here)
      anchor_ratios = C.anchor_box_ratios  # (3 in here)

      assert rpn_layer.shape[0] == 1

      (rows, cols) = rpn_layer.shape[1:3]

      curr_layer = 0

      # A.shape = (4, feature_map.height, feature_map.width, num_anchors)
      # Might be (4, 18, 25, 18) if resized image is 400 width and 300
      # A is the coordinates for 9 anchors for every point in the feature map
      # => all 18x25x9=4050 anchors cooridnates
      A = np.zeros((4, rpn_layer.shape[1], rpn_layer.shape[2], rpn_layer.shape[3]))

      for anchor_size in anchor_sizes:
            for anchor_ratio in anchor_ratios:
                  # anchor_x = (128 * 1) / 16 = 8  => width of current anchor
                  # anchor_y = (128 * 2) / 16 = 16 => height of current anchor
                  anchor_x = (anchor_size * anchor_ratio[0])/C.rpn_stride
                  anchor_y = (anchor_size * anchor_ratio[1])/C.rpn_stride
```

```
                # curr_layer: 0~8 (9 anchors)
                # the Kth anchor of all position in the feature map (9th in total)
                regr = regr_layer[0, :, :, 4 * curr_layer:4 * curr_layer + 4] # shape
=> (18, 25, 4)
                regr = np.transpose(regr, (2, 0, 1)) # shape => (4, 18, 25)

                # Create 18x25 mesh grid
                # For every point in x, there are all the y points and vice versa
                # X.shape = (18, 25)
                # Y.shape = (18, 25)
                X, Y = np.meshgrid(np.arange(cols),np. arange(rows))

                # Calculate anchor position and size for each feature map point
                A[0, :, :, curr_layer] = X - anchor_x/2 # Top left x coordinate
                A[1, :, :, curr_layer] = Y - anchor_y/2 # Top left y coordinate
                A[2, :, :, curr_layer] = anchor_x       # width of current anchor
                A[3, :, :, curr_layer] = anchor_y       # height of current anchor

                # Apply regression to x, y, w and h if there is rpn regression layer
                if use_regr:
                    A[:, :, :, curr_layer] = apply_regr_np(A[:, :, :, curr_layer],
regr)

                # Avoid width and height exceeding 1
                A[2, :, :, curr_layer] = np.maximum(1, A[2, :, :, curr_layer])
                A[3, :, :, curr_layer] = np.maximum(1, A[3, :, :, curr_layer])

                # Convert (x, y , w, h) to (x1, y1, x2, y2)
                # x1, y1 is top left coordinate
                # x2, y2 is bottom right coordinate
                A[2, :, :, curr_layer] += A[0, :, :, curr_layer]
                A[3, :, :, curr_layer] += A[1, :, :, curr_layer]

                # Avoid bboxes drawn outside the feature map
                A[0, :, :, curr_layer] = np.maximum(0, A[0, :, :, curr_layer])
                A[1, :, :, curr_layer] = np.maximum(0, A[1, :, :, curr_layer])
                A[2, :, :, curr_layer] = np.minimum(cols-1, A[2, :, :, curr_layer])
                A[3, :, :, curr_layer] = np.minimum(rows-1, A[3, :, :, curr_layer])

                curr_layer += 1

    all_boxes = np.reshape(A.transpose((0, 3, 1, 2)), (4, -1)).transpose((1, 0))  #
shape=(4050, 4)
    all_probs = rpn_layer.transpose((0, 3, 1, 2)).reshape((-1))                   #
shape=(4050,)

    x1 = all_boxes[:, 0]
    y1 = all_boxes[:, 1]
    x2 = all_boxes[:, 2]
    y2 = all_boxes[:, 3]

    # Find out the bboxes which is illegal and delete them from bboxes list
    idxs = np.where((x1 - x2 >= 0) | (y1 - y2 >= 0))

    all_boxes = np.delete(all_boxes, idxs, 0)
    all_probs = np.delete(all_probs, idxs, 0)

    # Apply non_max_suppression
    # Only extract the bboxes. Don't need rpn probs in the later process
    result = non_max_suppression_fast(all_boxes, all_probs,
overlap_thresh=overlap_thresh, max_boxes=max_boxes)[0]

    return result
```

```
# Transform train_on_batch return value
# to dict expected by on_batch_end callback
def named_logs(model, logs):
  result = {}
  for l in zip(model.metrics_names, logs):
    result[l[0]] = l[1]
  return result


#
#
# ---
#
#

#
#
# ---
#
#

# ### Start training



base_path = 'drive/My Drive/test_colab01'

train_path = 'drive/My Drive/test_colab01/train_labels.csv'

num_rois = 4 # Number of RoIs to process at once.

# Augmentation flag
horizontal_flips = True # Augment with horizontal flips in training.
vertical_flips = True   # Augment with vertical flips in training.
rot_90 = True           # Augment with 90 degree rotations in training.

output_weight_path = os.path.join(base_path, 'model_06/model_frcnn_vgg.hdf5')

record_path = os.path.join(base_path,'model_06/record.csv') # Record data (used to
save the losses, classification accuracy and mean average precision)

base_weight_path =
os.path.join(base_path,'model_06/vgg16_weights_tf_dim_ordering_tf_kernels.h5')

config_output_filename = os.path.join(base_path,'model_06/model_vgg_config.pickle')




# Create the config
C = Config()

C.use_horizontal_flips = horizontal_flips
C.use_vertical_flips = vertical_flips
C.rot_90 = rot_90

C.record_path = record_path
C.model_path = output_weight_path
C.num_rois = num_rois
```

```
C.base_net_weights = base_weight_path




#--------------------------------------------------------#
# This step will spend some time to load the data        #
#--------------------------------------------------------#
st = time.time()
train_imgs, classes_count, class_mapping = get_data(train_path)
print()
print('Spend %0.2f mins to load the data' % ((time.time()-st)/60) )




if 'bg' not in classes_count:
      classes_count['bg'] = 0
      class_mapping['bg'] = len(class_mapping)
# e.g.
#    classes_count: {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745, 'bg': 0}
#    class_mapping: {'Person': 0, 'Car': 1, 'Mobile phone': 2, 'bg': 3}
C.class_mapping = class_mapping

print('Training images per class:')
pprint.pprint(classes_count)
print('Num classes (including bg) = {}'.format(len(classes_count)))
print(class_mapping)

# Save the configuration
with open(config_output_filename, 'wb') as config_f:
      pickle.dump(C,config_f)
      print('Config has been written to {}, and can be loaded when testing to ensure
correct results'.format(config_output_filename))




# Shuffle the images with seed
random.seed(1)
random.shuffle(train_imgs)

print('Num train samples (images) {}'.format(len(train_imgs)))




# Get train data generator which generate X, Y, image_data
data_gen_train = get_anchor_gt(train_imgs, C, get_img_output_length, mode='train')


# #### Explore 'data_gen_train'
#
# data_gen_train is an **generator**, so we get the data by calling
**next(data_gen_train)**



from google.colab import drive
```

```
drive.mount('/content/drive')




X, Y, image_data, debug_img, debug_num_pos = next(data_gen_train)





print('Original image: height=%d width=%d'%(image_data['height'],
image_data['width']))
print('Resized image:  height=%d width=%d C.im_size=%d'%(X.shape[1], X.shape[2],
C.im_size))
print('Feature map size: height=%d width=%d C.rpn_stride=%d'%(Y[0].shape[1],
Y[0].shape[2], C.rpn_stride))
print(X.shape)
print(str(len(Y))+" includes 'y_rpn_cls' and 'y_rpn_regr'")
print('Shape of y_rpn_cls {}'.format(Y[0].shape))
print('Shape of y_rpn_regr {}'.format(Y[1].shape))
print(image_data)

print('Number of positive anchors for this image: %d' % (debug_num_pos))
if debug_num_pos==0:
    gt_x1, gt_x2 = image_data['bboxes'][0]['x1']*(X.shape[2]/image_data['height']),
image_data['bboxes'][0]['x2']*(X.shape[2]/image_data['height'])
    gt_y1, gt_y2 = image_data['bboxes'][0]['y1']*(X.shape[1]/image_data['width']),
image_data['bboxes'][0]['y2']*(X.shape[1]/image_data['width'])
    gt_x1, gt_y1, gt_x2, gt_y2 = int(gt_x1), int(gt_y1), int(gt_x2), int(gt_y2)

    img = debug_img.copy()
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    color = (0, 255, 0)
    cv2.putText(img, 'gt bbox', (gt_x1, gt_y1-5), cv2.FONT_HERSHEY_DUPLEX, 0.7, color,
1)
    cv2.rectangle(img, (gt_x1, gt_y1), (gt_x2, gt_y2), color, 2)
    cv2.circle(img, (int((gt_x1+gt_x2)/2), int((gt_y1+gt_y2)/2)), 3, color, -1)

    plt.grid()
    plt.imshow(img)
    plt.show()
else:
    cls = Y[0][0]
    pos_cls = np.where(cls==1)
    print(pos_cls)
    regr = Y[1][0]
    pos_regr = np.where(regr==1)
    print(pos_regr)
    print('y_rpn_cls for possible pos anchor:
{}'.format(cls[pos_cls[0][0],pos_cls[1][0],:]))
    print('y_rpn_regr for positive anchor:
{}'.format(regr[pos_regr[0][0],pos_regr[1][0],:]))

    gt_x1, gt_x2 = image_data['bboxes'][0]['x1']*(X.shape[2]/image_data['width']),
image_data['bboxes'][0]['x2']*(X.shape[2]/image_data['width'])
    gt_y1, gt_y2 = image_data['bboxes'][0]['y1']*(X.shape[1]/image_data['height']),
image_data['bboxes'][0]['y2']*(X.shape[1]/image_data['height'])
    gt_x1, gt_y1, gt_x2, gt_y2 = int(gt_x1), int(gt_y1), int(gt_x2), int(gt_y2)

    img = debug_img.copy()
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    color = (0, 255, 0)
```

```
    #   cv2.putText(img, 'gt bbox', (gt_x1, gt_y1-5), cv2.FONT_HERSHEY_DUPLEX, 0.7,
color, 1)
    cv2.rectangle(img, (gt_x1, gt_y1), (gt_x2, gt_y2), color, 2)
    cv2.circle(img, (int((gt_x1+gt_x2)/2), int((gt_y1+gt_y2)/2)), 3, color, -1)

    # Add text
    textLabel = 'gt bbox'
    (retval,baseLine) = cv2.getTextSize(textLabel,cv2.FONT_HERSHEY_COMPLEX,0.5,1)
    textOrg = (gt_x1, gt_y1+5)
    cv2.rectangle(img, (textOrg[0] - 5, textOrg[1]+baseLine - 5),
(textOrg[0]+retval[0] + 5, textOrg[1]-retval[1] - 5), (0, 0, 0), 2)
    cv2.rectangle(img, (textOrg[0] - 5,textOrg[1]+baseLine - 5), (textOrg[0]+retval[0]
+ 5, textOrg[1]-retval[1] - 5), (255, 255, 255), -1)
    cv2.putText(img, textLabel, textOrg, cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 0, 0), 1)

    # Draw positive anchors according to the y_rpn_regr
    for i in range(debug_num_pos):

        color = (100+i*(155/4), 0, 100+i*(155/4))

        idx = pos_regr[2][i*4]/4
        print(idx)
        anchor_size = C.anchor_box_scales[int(idx/4)]
        anchor_ratio = C.anchor_box_ratios[2-int((idx+1)%3)]

        center = (pos_regr[1][i*4]*C.rpn_stride, pos_regr[0][i*4]*C.rpn_stride)
        print('Center position of positive anchor: ', center)
        cv2.circle(img, center, 3, color, -1)
        anc_w, anc_h = anchor_size*anchor_ratio[0], anchor_size*anchor_ratio[1]
        cv2.rectangle(img, (center[0]-int(anc_w/2), center[1]-int(anc_h/2)),
(center[0]+int(anc_w/2), center[1]+int(anc_h/2)), color, 2)
#         cv2.putText(img, 'pos anchor bbox '+str(i+1), (center[0]-int(anc_w/2),
center[1]-int(anc_h/2)-5), cv2.FONT_HERSHEY_DUPLEX, 0.5, color, 1)

print('Green bboxes is ground-truth bbox. Others are positive anchors')
plt.figure(figsize=(8,8))
plt.grid()
plt.imshow(img)
plt.show()


# #### Build the model




input_shape_img = (None, None, 3)

img_input = Input(shape=input_shape_img)
roi_input = Input(shape=(None, 4))

# define the base network (VGG here, can be Resnet50, Inception, etc)
shared_layers = nn_base(img_input, trainable=True)





# define the RPN, built on the base layers
num_anchors = len(C.anchor_box_scales) * len(C.anchor_box_ratios) # 9
rpn = rpn_layer(shared_layers, num_anchors)

classifier = classifier_layer(shared_layers, roi_input, C.num_rois,
nb_classes=len(classes_count))
```

```
model_rpn = Model(img_input, rpn[:2])
model_classifier = Model([img_input, roi_input], classifier)

# this is a model that holds both the RPN and the classifier, used to load/save
weights for the models
model_all = Model([img_input, roi_input], rpn[:2] + classifier)

# Because the google colab can only run the session several hours one time (then you
need to connect again),
# we need to save the model and load the model to continue training
if not os.path.isfile(C.model_path):
    #If this is the begin of the training, load the pre-traind base network such as
vgg-16
    try:
        print('This is the first time of your training')
        print('loading weights from {}'.format(C.base_net_weights))
        model_rpn.load_weights(C.base_net_weights, by_name=True)
        model_classifier.load_weights(C.base_net_weights, by_name=True)
    except:
        print('Could not load pretrained model weights. Weights can be found in the
keras application folder
https://github.com/fchollet/keras/tree/master/keras/applications')

    # Create the record.csv file to record losses, acc and mAP
    record_df = pd.DataFrame(columns=['mean_overlapping_bboxes', 'class_acc',
'loss_rpn_cls', 'loss_rpn_regr', 'loss_class_cls', 'loss_class_regr', 'curr_loss',
'elapsed_time', 'mAP'])
else:
    # If this is a continued training, load the trained model from before
    print('Continue training based on previous trained model')
    print('Loading weights from {}'.format(C.model_path))
    model_rpn.load_weights(C.model_path, by_name=True)
    model_classifier.load_weights(C.model_path, by_name=True)

    # Load the records
    record_df = pd.read_csv(record_path)

    r_mean_overlapping_bboxes = record_df['mean_overlapping_bboxes']
    r_class_acc = record_df['class_acc']
    r_loss_rpn_cls = record_df['loss_rpn_cls']
    r_loss_rpn_regr = record_df['loss_rpn_regr']
    r_loss_class_cls = record_df['loss_class_cls']
    r_loss_class_regr = record_df['loss_class_regr']
    r_curr_loss = record_df['curr_loss']
    r_elapsed_time = record_df['elapsed_time']
    r_mAP = record_df['mAP']

    print('Already train %dK batches'% (len(record_df)))




optimizer = Adam(lr=1e-5)
optimizer_classifier = Adam(lr=1e-5)
model_rpn.compile(optimizer=optimizer, loss=[rpn_loss_cls(num_anchors),
rpn_loss_regr(num_anchors)])
model_classifier.compile(optimizer=optimizer_classifier, loss=[class_loss_cls,
class_loss_regr(len(classes_count)-1)],
metrics={'dense_class_{}'.format(len(classes_count)): 'accuracy'})
model_all.compile(optimizer='sgd', loss='mae')
```

```
# Training setting
total_epochs = len(record_df)
r_epochs = len(record_df)

epoch_length = 80
num_epochs = 32
iter_num = 0

total_epochs += num_epochs

losses = np.zeros((epoch_length, 5))
rpn_accuracy_rpn_monitor = []
rpn_accuracy_for_epoch = []

if len(record_df)==0:
    best_loss = np.Inf
else:
    best_loss = np.min(r_curr_loss)




print(len(record_df))




start_time = time.time()
for epoch_num in range(num_epochs):

    progbar = generic_utils.Progbar(epoch_length)
    print('Epoch {}/{}'.format(r_epochs + 1, total_epochs))

    r_epochs += 1

    while True:
        try:

            if len(rpn_accuracy_rpn_monitor) == epoch_length and C.verbose:
                mean_overlapping_bboxes =
float(sum(rpn_accuracy_rpn_monitor))/len(rpn_accuracy_rpn_monitor)
                rpn_accuracy_rpn_monitor = []
#                 print('Average number of overlapping bounding boxes from RPN = {}
for {} previous iterations'.format(mean_overlapping_bboxes, epoch_length))
                if mean_overlapping_bboxes == 0:
                    print('RPN is not producing bounding boxes that overlap the ground
truth boxes. Check RPN settings or keep training.')

            # Generate X (x_img) and label Y ([y_rpn_cls, y_rpn_regr])
            X, Y, img_data, debug_img, debug_num_pos = next(data_gen_train)

            # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
            loss_rpn = model_rpn.train_on_batch(X, Y)

            # Get predicted rpn from rpn model [rpn_cls, rpn_regr]
            P_rpn = model_rpn.predict_on_batch(X)

            # R: bboxes (shape=(300,4))
            # Convert rpn layer to roi bboxes
            #image_dim_ordering deprecated use image_data_format
            R = rpn_to_roi(P_rpn[0], P_rpn[1], C, K.image_data_format(),
use_regr=True, overlap_thresh=0.7, max_boxes=300)
```

```
            # note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
            # X2: bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300
non_max_suppression bboxes
            # Y1: one hot code for bboxes from above => x_roi (X)
            # Y2: corresponding labels and corresponding gt bboxes
            X2, Y1, Y2, IouS = calc_iou(R, img_data, C, class_mapping)

            # If X2 is None means there are no matching bboxes
            if X2 is None:
                rpn_accuracy_rpn_monitor.append(0)
                rpn_accuracy_for_epoch.append(0)
                continue

            # Find out the positive anchors and negative anchors
            neg_samples = np.where(Y1[0, :, -1] == 1)
            pos_samples = np.where(Y1[0, :, -1] == 0)

            if len(neg_samples) > 0:
                neg_samples = neg_samples[0]
            else:
                neg_samples = []

            if len(pos_samples) > 0:
                pos_samples = pos_samples[0]
            else:
                pos_samples = []

            rpn_accuracy_rpn_monitor.append(len(pos_samples))
            rpn_accuracy_for_epoch.append((len(pos_samples)))

            if C.num_rois > 1:
                # If number of positive anchors is larger than 4//2 = 2, randomly
choose 2 pos samples
                if len(pos_samples) < C.num_rois//2:
                    selected_pos_samples = pos_samples.tolist()
                else:
                    selected_pos_samples = np.random.choice(pos_samples,
C.num_rois//2, replace=False).tolist()

                # Randomly choose (num_rois - num_pos) neg samples
                try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois -
len(selected_pos_samples), replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois -
len(selected_pos_samples), replace=True).tolist()

                # Save all the pos and neg samples in sel_samples
                sel_samples = selected_pos_samples + selected_neg_samples
            else:
                # in the extreme case where num_rois = 1, we pick a random pos or neg
sample
                selected_pos_samples = pos_samples.tolist()
                selected_neg_samples = neg_samples.tolist()
                if np.random.randint(0, 2):
                    sel_samples = random.choice(neg_samples)
                else:
                    sel_samples = random.choice(pos_samples)

            # training_data: [X, X2[:, sel_samples, :]]
            # labels: [Y1[:, sel_samples, :], Y2[:, sel_samples, :]]
            #  X                     => img_data resized image
            #  X2[:, sel_samples, :] => num_rois (4 in here) bboxes which contains
selected neg and pos
```

```
            #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which
contains selected neg and pos
            #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which
contains selected neg and pos
            loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]],
[Y1[:, sel_samples, :], Y2[:, sel_samples, :]])



            losses[iter_num, 0] = loss_rpn[1]
            losses[iter_num, 1] = loss_rpn[2]

            losses[iter_num, 2] = loss_class[1]
            losses[iter_num, 3] = loss_class[2]
            losses[iter_num, 4] = loss_class[3]
            iter_num += 1

            progbar.update(iter_num, [('rpn_cls', np.mean(losses[:iter_num, 0])),
('rpn_regr', np.mean(losses[:iter_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])),
('final_regr', np.mean(losses[:iter_num, 3]))])

            if iter_num == epoch_length:
                loss_rpn_cls = np.mean(losses[:, 0])
                loss_rpn_regr = np.mean(losses[:, 1])
                loss_class_cls = np.mean(losses[:, 2])
                loss_class_regr = np.mean(losses[:, 3])
                class_acc = np.mean(losses[:, 4])

                mean_overlapping_bboxes = float(sum(rpn_accuracy_for_epoch)) /
len(rpn_accuracy_for_epoch)
                rpn_accuracy_for_epoch = []

                if C.verbose:
                    print('Mean number of bounding boxes from RPN overlapping ground
truth boxes: {}'.format(mean_overlapping_bboxes))
                    print('Classifier accuracy for bounding boxes from RPN:
{}'.format(class_acc))
                    print('Loss RPN classifier: {}'.format(loss_rpn_cls))
                    print('Loss RPN regression: {}'.format(loss_rpn_regr))
                    print('Loss Detector classifier: {}'.format(loss_class_cls))
                    print('Loss Detector regression: {}'.format(loss_class_regr))
                    print('Total loss: {}'.format(loss_rpn_cls + loss_rpn_regr +
loss_class_cls + loss_class_regr))
                    print('Elapsed time: {}'.format(time.time() - start_time))
                    elapsed_time = (time.time()-start_time)/60

                curr_loss = loss_rpn_cls + loss_rpn_regr + loss_class_cls +
loss_class_regr
                iter_num = 0
                start_time = time.time()

                if curr_loss < best_loss:
                    if C.verbose:
                        print('Total loss decreased from {} to {}, saving
weights'.format(best_loss,curr_loss))
                    best_loss = curr_loss
                    model_all.save_weights(C.model_path)

                new_row = {'mean_overlapping_bboxes':round(mean_overlapping_bboxes,
3),
                           'class_acc':round(class_acc, 3),
                           'loss_rpn_cls':round(loss_rpn_cls, 3),
                           'loss_rpn_regr':round(loss_rpn_regr, 3),
                           'loss_class_cls':round(loss_class_cls, 3),
```

```
                              'loss_class_regr':round(loss_class_regr, 3),
                              'curr_loss':round(curr_loss, 3),
                              'elapsed_time':round(elapsed_time, 3),
                              'mAP': 0}

                record_df = record_df.append(new_row, ignore_index=True)
                record_df.to_csv(record_path, index=0)

                break

        except Exception as e:
            print('Exception: {}'.format(e))
            continue

print('Training complete, exiting.')




plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['mean_overlapping_bboxes'], 'r')
plt.title('mean_overlapping_bboxes')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['class_acc'], 'r')
plt.title('class_acc')

plt.show()

plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_cls'], 'r')
plt.title('loss_rpn_cls')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_regr'], 'r')
plt.title('loss_rpn_regr')
plt.show()


plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['loss_class_cls'], 'r')
plt.title('loss_class_cls')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['loss_class_regr'], 'r')
plt.title('loss_class_regr')
plt.show()

plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'r')
plt.title('total_loss')
plt.show()

# plt.figure(figsize=(15,5))
# plt.subplot(1,2,1)
# plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'r')
# plt.title('total_loss')
# plt.subplot(1,2,2)
# plt.plot(np.arange(0, r_epochs), record_df['elapsed_time'], 'r')
# plt.title('elapsed_time')
# plt.show()

# plt.title('loss')
# plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_cls'], 'b')
# plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_regr'], 'g')
```

```
# plt.plot(np.arange(0, r_epochs), record_df['loss_class_cls'], 'r')
# plt.plot(np.arange(0, r_epochs), record_df['loss_class_regr'], 'c')
# # plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'm')
# plt.show()
```