AFRICAN UNIVERSITY OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Masters Thesis

Fast and Accurate Feature-based Region Identification

Abstract

There have been several improvement in object detection and semantic segmentation results in recent years. Baseline systems that drives these advances are Fast/Faster R-CNN, Fully Convolutional Network and recently **Mask R-CNN** and its variant that has a weight transfer function. Mask R-CNN is the state-of-art. This research extends the application of the state-of-art in object detection and semantic segmentation in drone based datasets.

Existing drone datasets was used to learn semantic segmentation on drone images using **Mask R-CNN**. And a new drone dataset will be collected, labelled, annotated with a bounding box object detection.

Drone based images will be collected with the drone developed by the Robotic team at African University of Science and Technology, Abuja, which will be made public for academic researches.
This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

JUNE 2019 MADUAKOR FRANCIS

THESIS SUPERVISOR:
PROF. DR. LEHEL CSATÓ

FACULTY OF MATHEMATICS AND INFORMATICS,
BABEŞ BOLYAI UNIVERSITY OF CLUJ-NAPOCA,
ROMANIA

AFRICAN UNIVERSITY OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Masters Thesis

Fast and Accurate Feature-based Region Identification

THESIS SUPERVISOR:

PROF. DR. LEHEL CSATÓ

FACULTY OF MATHEMATICS AND INFORMATICS,
BABEŞ BOLYAI UNIVERSITY OF CLUJ-NAPOCA,
ROMANIA

STUDENT:

MADUAKOR FRANCIS

JUNE 2019

Contents

List of Figures

List of Algorithms

1. Chapter

Introduction

1.1 Introduction

Images and videos are collected everyday by different sources. Recognizing objects, segmenting localizing and classifying them has been a major area of interest in computer vision. Significant progress has been made commencing from use of low-level image features, such as **scale invariant feature transform** SIFT [A] and **histogram of oriented gradients HOG** [B] , in sophisticated machine learning frameworks to the use of multi-layer convolutional networks to compute highly discriminative, and invariant features [C]. SIFT and HOG are feature descriptor and semi-local orientation histograms that counts occurrences of gradient orientation in localized portions of an image. Just as Convolutional Neural Network (CNN) is traced to the Fukushima's **"neocognitron"** [D], a hierarchical and shift-invariant model for pattern recognition, the use of CNN for region-based identification (R-CNN)[C] can also be traced back to the same. After CNN was considered inelegant in the 1990s due to the rise of support vector machine (SVM), in 2012 it was revitalize by Krizhevsky et al. [D] by demonstrating a valuable improvement in image classification accuracy on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [E] and included new mechanisms to CNN like rectified linear unit (ReLU) and, dropout regularization. To perform object detection with CNN and in attempt to bridge the gap between image segmentation and object detection two issues were fixed by R.Girshick et al [C]. First was the localization of objects with a Deep Network and training a high-capacity model with only a small quantity of annotated detection data. Use of a sliding-window detector was proposed for the localization of object but was not preferred because it can only work for one object detection and all object in an image has to have a common aspect ratio for its use in multiple object detection. Instead the localization problem was solved by operating within the "recognition using regions" paradigm.

Fast R-CNN was introduced in 2015 by Girshick [F]. A single-stage training algorithm that jointly learns to classify object proposals and refine their spatial locations was demonstrated. This tackled the problem of complexity that arises in other deep ConvNets [D,G, H], caused by the multi-stage pipelines that are slow. The slow nature is due to the fact that detection requires accurate localization of objects that creates the challenge of that many proposals (candidate object locations) must be processed and these proposals provides only rough localization that must be refined to achieve precise localization. Fast R-CNN is 9 X faster than R-CNN [C] and 3 X faster than SPPnet [I]. R-CNN was speed up by **Spatial pyramid pooling networks (SPPnets)**[I] by sharing computation. A convolutional feature map

for the entire input image was computed by SPPnet method. After which it then classifies each object proposal using a feature vector extracted from the shared feature map. SPPnet also has obvious pitfalls. It is a multi-stage pipeline similarly to R-CNN that involves extracting features, refining a network with log loss, training SVMs, and lastly fitting bounding-box regressors. Features are also written to disk. But unlike R-CNN, the refining algorithm demonstrated in SPPnet cannot update the convolutional layers that precede the spatial pyramid pooling. This constraint limits the accuracy of very deep networks. Additional efforts were made to reduce the running time of deep ConvNets for object detection and segmentation. Regional proposal computation is the root of this expensive running time in detection networks. A fully convolutional network that simultaneously predicts object bounds and objectness scores at each position called **Region Proposal Network (RPN)** was developed by Ren et al [J]. RPN shares full-image convolutional features with the detection network, thus permitting virtually cost-free region proposals and it is trained end-to-end to generate high-quality region proposals. Integrating RPN and Fast R-CNN into a unit network by sharing their convolutional features results to Faster R-CNN. Anchor boxes that acts as reference at multiple scales and aspect ratios were introduced in Faster R-CNN instead of the pyramids of filters used in earlier methods. RPNs are developed to coherently speculate region proposals with an extensive range of scales and aspect ratios. Changing the architecture of the pyramids of filter to a top-down architecture with lateral connections improved the efficiency of this pyramids [K]. This is applied in building high-level semantic feature maps at all scales. This new architecture is called **Feature Pyramid Network (FPN)** [K]. In various applications and uses it displayed a notable improvement as a generic feature extractor. When used in a Faster R-CNN it achieved results that supersedes that of Faster R-CNN alone. In order to generate a high-quality segmentation mask for object instances in an image, Mask R-CNN was developed [L]. Mask R-CNN add another branch to the Faster R-CNN. In addition to the bounding box recognition system a branch for predicting an object mask in parallel was added. It affixes only a bijou overhead to Faster R-CNN, running at **5 fps**.

The accessibility and use of drone technology is at the increase currently. It is tackling challenges in various spheres and areas like defence, shipping of consumer goods, disease controls, events coverage and so on. One of the most important application of drone is for collection of images and videos. These data collected can be used for different purposes. This work will extend the state-of-the-art Mask R-CNN for segmentation of objects in image instances collected by a drone. It detects about 22 classes including tree, grass, other vegetation, dirt, grave, rocks, water, paved area, pool, person, dog, car, bicycle, roof, wall, fence, fence-pole, window, door, and obstacle. For the training of the model high resolution images at 1Hz with pixel-accurate annotation was used.

In Chapter 2 of this work will discuss theory of CNN and Mask RCNN deeply. The first part will discuss the backbone of Mask RCNN, followed by Regional Proposal Network, ROI Classifier and Bounding Box Regressor and lastly Segmentation Mask. Chapter 3 will discuss fully segmentation on drone dataset. Chapter 4 will explore the methodology and implementation of the work.

2. Chapter

Chapter about theory

2.1 Computer Vision Tasks

With the advance of computer vision, task in computer vision has moved from simple tasks of image classifaication to complex task like semantic and instance segmentation. Deep learning has made this possible, especially using the Vo

2.1.1 Image Classification

Image classification is the process of assigning land cover classes to pixels. Image classification refers to the task of extracting information classes from a multiband raster image. The resulting raster from image classification can be used to create thematic maps. Depending on the interaction between the analyst and the computer during classification, there are two types of classification: supervised and unsupervised. The image classification plays an important role in environmental and socioeconomic applications. In order to improve the classification accuracy, scientists have laid path in developing the advanced classification techniques. Image classification analyzes the numerical properties of various image features and organizes data into categories. Classification algorithms typically employ two phases of processing: training and testing. In the initial training phase, characteristic properties of typical image features are isolated and, based on these, a unique description of each classification category, i.e. training class, is created. In the subsequent testing phase, these feature-space partitions are used to classify image features. The description of training classes is an extremely important component of the classification process. In supervised classification, statistical processes (i.e. based on an a priori knowledge of probability distribution functions) or distribution-free processes can be used to extract class descriptors. Unsupervised classification relies on clustering algorithms to automatically segment the training data into prototype classes. In either case, the motivating criteria for constructing training classes are that they are:

1. Independent, e.a change in the description of one training class should not change the value of another,

2. Discriminatory, e.different image features should have significantly different descriptions, and

3. Reliable, all image features within a training group should share the common definitive descriptions of that group.

This representation allows us to consider each image feature as occupying a point, and each training class as occupying a sub-space (i.e. a representative point surrounded by some spread, or deviation), within the n-dimensional classification space. Viewed as such, the classification problem is that of determining to which sub-space class each feature vector belongs.

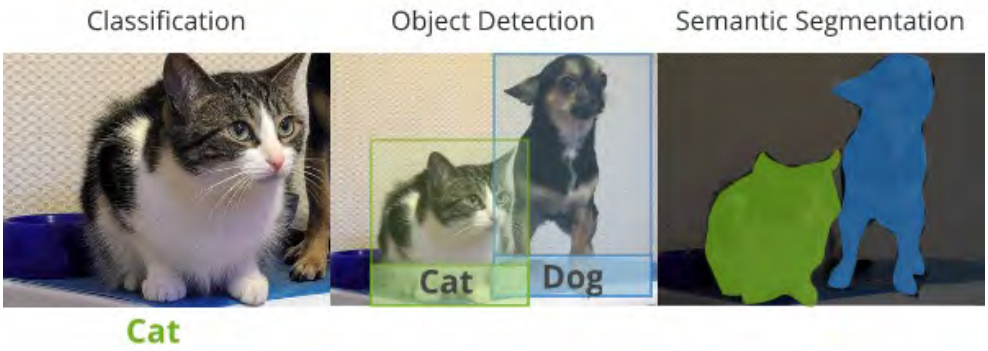

Figure 2.1: Image Classification,Object detection Semantic Segmentation

2.1.2 Object Detection

The goal of object detection is to detect all instances of objects from a known class, such as people, cars or faces in an image. Typically only a small number of instances of the object are present in the image, but there is a very large number of possible locations and scales at which they can occur and that need to somehow be explored. Each detection is reported with some form of pose information. This could be as simple as the location of the object, a location and scale, or the extent of the object defined in terms of a bounding box. In other situations the pose information is more detailed and contains the parameters of a linear or non-linear transformation. For example a face detector may compute the locations of the eyes, nose and mouth, in addition to the bounding box of the face. An example of a vehicle and person detection that specifies the locations of certain parts is shown in Figure 1. The pose could also be defined by a three-dimensional transformation specifying the location of the object relative to the camera. Object detection systems construct a model for an object class from a set of training examples. In the case of a fixed rigid object only one example may be needed, but more generally multiple training examples are necessary to capture certain aspects of class variability.

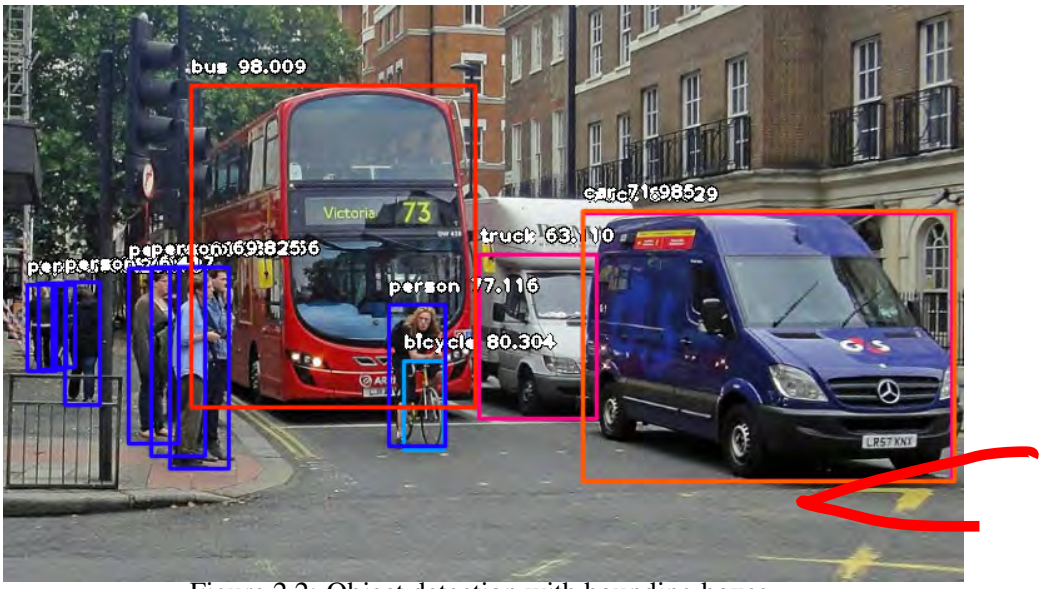

Figure 2.2: Object detection with bounding boxes.

Object detection methods fall into two major categories, generative and discriminative. The first consists of a probability model for the pose variability of the objects together with an appearance model: a probability model for the image appearance conditional on a given pose, together with a model for background, i.e. non-object images. The model parameters can be estimated from training data and the decisions are based on ratios of posterior probabilities. The second typically builds a classifier that can discriminate between images (or sub-images) containing the object and those not containing the object. The parameters of the classifier are selected to minimize mistakes on the training data, often with a regularization bias to avoid overfitting. Other distinctions among detection algorithms have to do with the computational tools used to scan the entire image or search over possible poses, the type of image representation with which the models are constructed, and what type and how much training data is required to build a model.

2.1.3 Semantic Segmentation

Segmentation is essential for image analysis tasks. Semantic segmentation describes the process of associating each pixel of an image with a class label, (such as flower, person, road, sky, ocean, or car). Semantic image segmentation can be applied effectively to any task that involves the segmentation of visual information. Examples include road segmentation for autonomous vehicles, medical image segmentation, scene segmentation for robot perception, and in image editing tools. Whilst currently available systems provide accurate object recognition, they are unable to delineate the boundaries between objects with the same accuracy.

Oxford researchers have developed a novel neural network component for semantic segmentation that enhances the ability to recognise and delineate objects. This invention can be applied to improve any situation requiring the segmentation of visual information.

Semantic image segmentation plays a crucial role in image understanding, allowing a computer to recognise objects in images. Recognition and delineation of objects is achieved through classification of each pixel in an image. Such processes have a wide range of applications in computer vision, in diverse and growing fields such as vehicle autonomy and medical imaging.

The previous state-of-the-art image segmentation systems used Fully Convolutional Neural Network (FCNN) components, which offer excellent accuracy in recognising objects. Whilst this development represented a significant improvement in semantic segmentation, these networks do not perform well in delineating object boundaries. Conditional Random Fields (CRFs) can be employed in a post-processing step to improve object boundary delineation, however, this is not an optimum solution owing to a lack of integration with the deep network.

Figure 2.3: Image with semantic segmentation.

Oxford researchers have developed a neural network component for semantic segmentation that harnesses the exceptional object recognition of FCNNs and the powerful boundary delineation of CRFs. CRFs are fully integrated as recurrent neural networks, resulting in a system that offers enhanced performance compared to the previous state-of-the-art. The novel system can be applied to any task that involves the segmentation of visual information. Examples include road segmentation for autonomous vehicles, medical image segmentation, scene segmentation for robot perception, and in image editing tools. Oxford University Innovation is seeking industrial partners that wish to explore the use of this system for commercial applications.

2.1.4 Instance Segmentation

Instance segmentation is one step ahead of semantic segmentation wherein along with pixel level classification, we expect the computer to classify each instance of a class separately. For example in the image above there are 3 people, technically 3 instances of the class “Person”. All the 3 are classified

separately (in a different color). But semantic segmentation does not differentiate between the instances of a particular class.

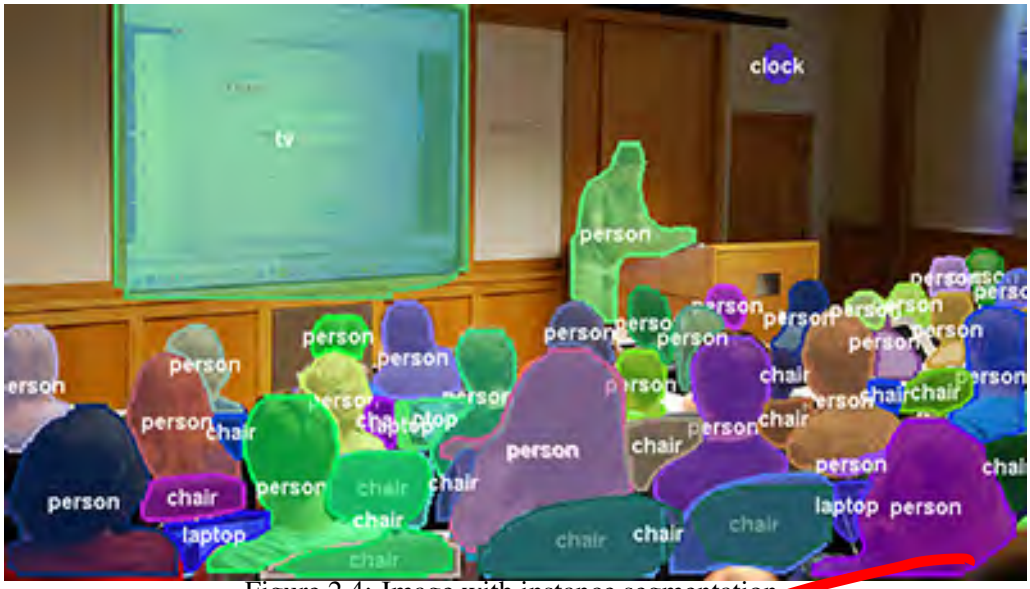

Figure 2.4: Image with instance segmentation

2.2 CNN for Object Detection and Segmentation

2.2.1 Backbone

Residual Networks (RESNET)

ResNet is an essential neural network that serves as a backbone to Mask R-CNN and numerous computer vision tasks. It makes the training of extremely deep neural networks possible which was very difficult before then due to the challenge of vanishing gradients, that hampers convergence in the network. According to [M] before RESNET, the problem of *vanishing gradient* has been mainly addressed by normalized initialization [N, O, P, Q] and intermediate normalization layers [R], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with backpropagation [S]. Looking at the sample scenario of the vanishing gradients or degradation. A worst-case scenario of vanishing gradient is the case was the early layers of a deeper model can be replaced with a shallow network and the other layers can act as an identity function. The shallow network and its deeper counterpart give the same accuracy. So deeper models do not perform well due to degradation. When a deeper network is used it approximates the mapping than its shallower variant and decreases the error by a notable margin. Also, the deeper network had issues of degradation. To solve this problem ResNet introduced the concept of skip connection.

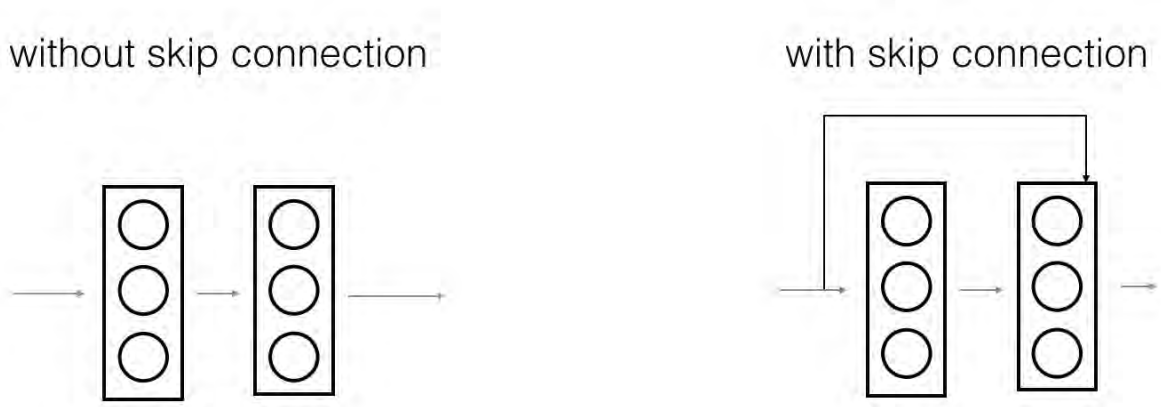

Figure 2.5: Skip connection of ResNet.

Without a skip connection, deep convolution networks are stacked together one after the other. With a skip connection deep convolution networks are tacked together but this time the original input is added to the output of the convolution block. Mathematically representing this, we can consider a mapping or space G(x) to be fitted by some stacked layers of an entire network. X denotes the inputs in the first layer of the net. This layer will approximate a residual function Z(x) = G(x)-x by hypothesizing. Therefore the original function or mapping G(x) becomes Z(x) + x. The input and output dimensions are expected to be of the same dimension for this work properly. It is worth noting that ResNet, contained 152 layers, won ILSVRC 2015 with an error rate of 3.6 percent beating even humans with their error rate of circa 5 – 10 percent, and replacing VGG-16 layers in Faster R-CNN with ResNet-101 produced relative improvements of 28 percent. It also trained networks with 100 layers and 1000 layers.

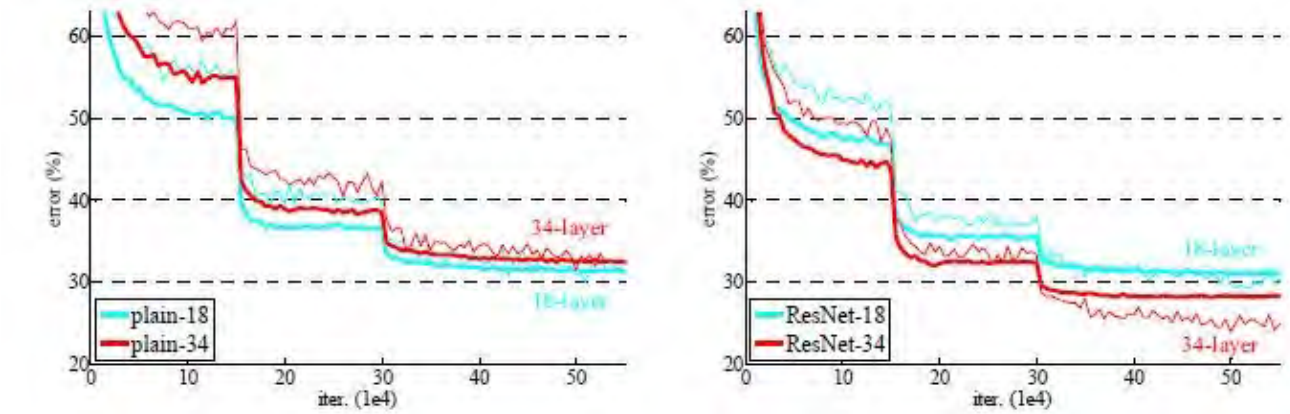

Figure 2.6: Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts [M].

Feature Pyramid Network

Feature Pyramid Network (FPN) is a generic feature extractor used in various application for recognizing objects at different scales, developed by Lin et al [K]. For the recognition system for detecting objects

at various scales, feature pyramid is a primary constituent of such a system. Pyramid representation has the problem of computing and memory intensiveness and has been avoided in the deep learning object detectors. Feature Pyramid Network (FPN) solves this problem by restructuring the architecture of the pyramid to a top-down architecture with lateral connections. The multi-scale, pyramidal hierarchy of deep ConvNet was leveraged to develop Feature Pyramid Network (FPN). Pyramids of FPN are scale-invariant. This means that when an object scale changes it is offset by shifting its level in the pyramid. Before the introduction of FPN, some ways were used in the extraction of features from images. Initially, hand-engineered features [U] were used and it makes use of featurized image pyramids. ConvNet is more robust to variance in scale, capable of representing higher-level semantics, and so features from it have quickly replaced engineered features. According to [L] this ConvNets gives multi-scale feature representation in which all levels are semantically strong, including the high-resolution levels. Featuring each level of an image pyramid comes with the profound limitation of increase in inference time which makes it impractical for real applications. FPN explored the pyramidal shape of a ConvNet's feature hierarchy to build a feature pyramid that has strong features with high-resolution at all scales. This gives a feature pyramid that has profound semantics at all phases and is constructed quickly from a unit input image scale

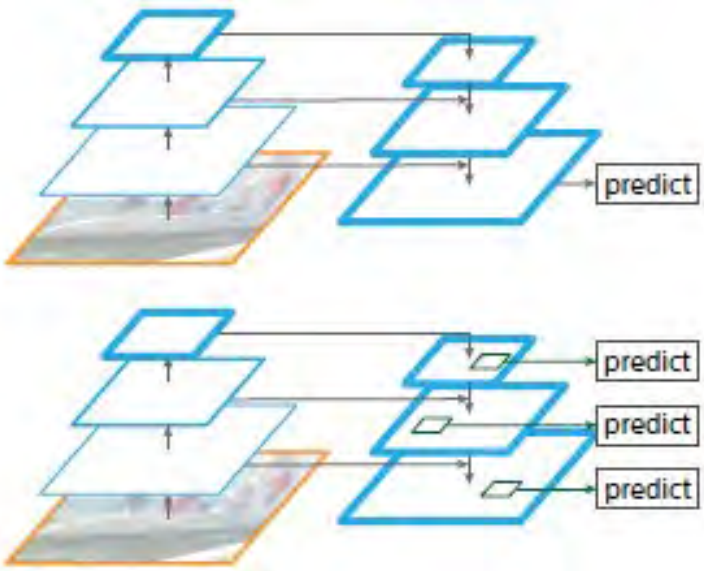

Figure 2.7: Top: a top-down architecture with skip connections, where predictions are made on the finest level (e.g., [T]). Bottom: FPN model that has a similar structure but leverages it as a feature pyramid, with predictions made independently at all levels [K].

FPN was applied in Regional Proposal Network (RPN) and Fast R-CNN. With the new adaptations, RPN could be naturally trained and tested with FPN, Using FPN in a basic Faster R-CNN system, the result surpasses all existing single-model entries including those from the COCO 2016 challenge winners.

2.2.2 Regional Proposal Network

Regional Proposal Network (RPN) is a useful network effectively used in R-CNN that scans the image in a sliding window pattern over the anchors. It proposes multiple objects that are recognizable in a particular image. The last convolutional layer that is produced by the Faster R-CNN is called the feature map. A proposal is generated for the region where the object lies by sliding over a feature map a network, which is the RPN. RPN suggest where an object lies in an image.

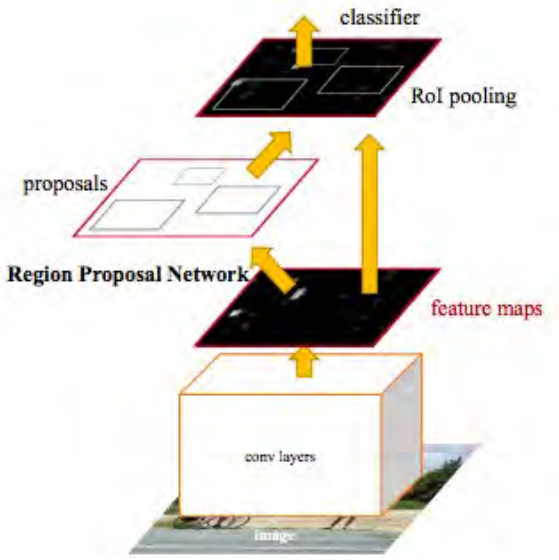

Figure 2.8: The architecture of Faster R-CNN. RPN generate the proposal for the objects. [ ].

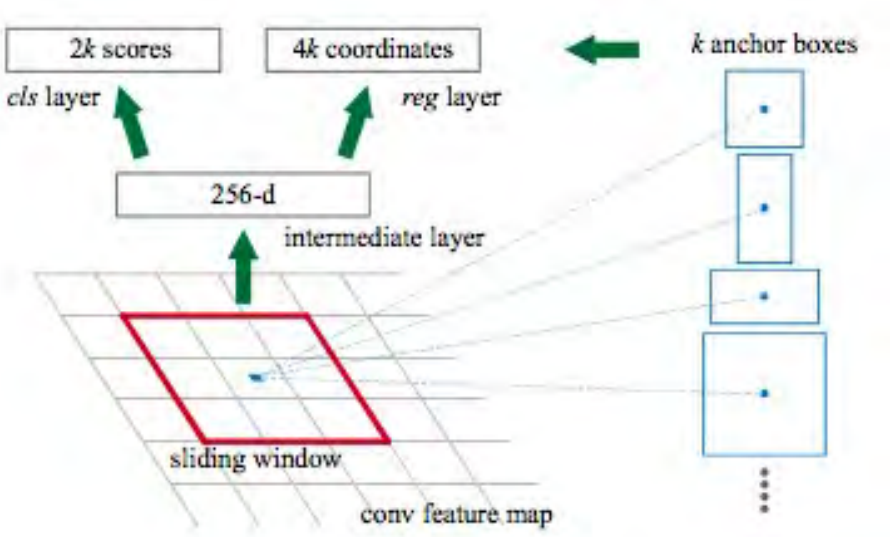

Figure 2.9: RPN Architecture [J].

Analyzing the architecture of RPN, the intermediate layer divides into a classifier and regressor layers, and the concept of the anchor was introduced. Anchor are boxes of different sizes and aspect

ratio that are generated over an image that determines the ideal location, shape, and size of objects in the image. They overlap to fill up as much of the image as possible. Thousands of anchor boxes are generated for this. For each anchor box, the object's bounding box that has the highest overlap is divided by non-overlap. This is termed Intersection Over Union (IOU). If the highest IOU is greater than 50 percent, the anchor box determines the object that gave the highest IOU. But if it is greater than 40 percent the true detection is ambiguous, and if it is less than 40 percent it predicts no object. Classifier gives the probability of a proposal containing the target object. Regression regresses the coordinates of the proposals. RPN is also used in Mask RCNN.

Figure 2.10: Anchor boxes (dotted) and the shift/scale applied to them to fit the object precisely (solid). Several anchors can map to the same object.

The RPN produces two results for each anchor; Anchor Class i.e. either the foreground or the background, and Bounding Box Refinement, an estimation to rectify the anchor box that fits the object well. This is a change in x,y, width, height.

2.2.3 ROI Classifier and Bounding Box Regressor

Region of Interest (ROI) classifier is proposed by the Region Proposal Network (RPN) and similarly, like the RPN, it produces two results for each ROI. First, the *class*, it produces the classes of the object in the ROI, but it is deeper and can classify regions to specific classes (car, person, nucleus, etc). And secondly the *Bounding Box Refinement*, which further refines the location and size of the bounding box to envelope the object.

ROI Pooling

Input sizes of a classifier vary. Classifiers require fixed, stable input size and can't manage varying input sizes. Bounding box refinement in RPN produces ROI boxes of various sizes. ROI Pooling tackles this

challenge. Cropping a part of a feature map and resizing it to a fixed size is termed ROI pooling. It is very much alike to the concept of resizing a cropped image.

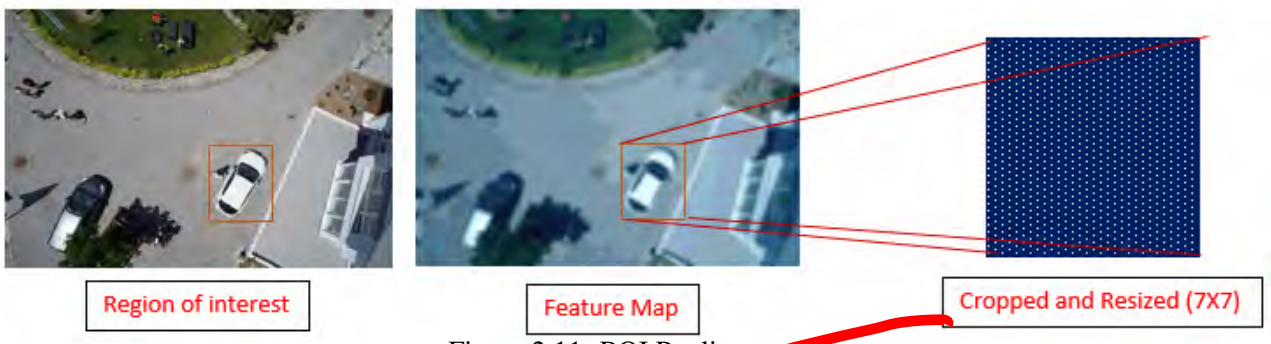

Figure 2.11: ROI Pooling

2.2.4 Segmentation Mask

Mask RCNN further added an extra branch to what Faster RCNN has. This is the mask branch. The Mask branch is a convolutional network that receives the positive regions selected by the ROI classifier and builds low resolution masks for them. The resolution is about 28x 28 pixels. They are soft masks, constituted by float numbers, so they contain more ingredients than binary masks. The size of the mask enables the mask branch to be light. When training the model, the ground-truth masks is scaled down to 28x28 to compute the loss, and when inferencing the predicted masks is scaled up to the size of the ROI bounding box and that produces the final masks, one per object.

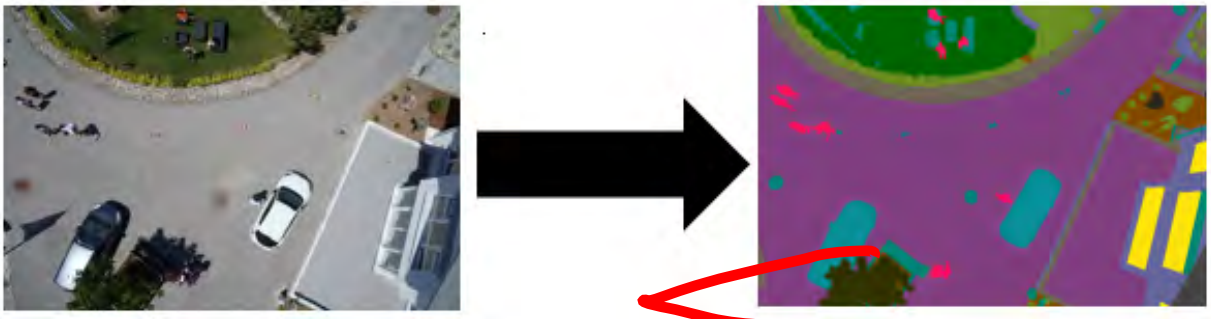

Figure 2.12: Segmentation Mask of a drone-based imageset .

2.3 Drone-Based Dataset

Drones (or UAVs) furnished with high resolution cameras have been utilized in a wide range of applications, including agricultural, aerial photography, fast delivery, surveillance, etc. This has made, automatic comprehension of visual data collected from drones become highly demanding, which brings computer vision to drones more and more closely. Outstanding advancements have been made in general computer vision algorithms, such as detection and tracking, yet these algorithms are not usually flawless for dealing with sequences or images captured by drones, due to various difficulties such as view point changes and

scales. Consequently, developing and appraising new vision algorithms for drone generated visual data is a key problem in drone-based applications. The major challenge of segmentation in drone generated visual data is the lack of proper datasets for these. A VisDrone Dataset was produced in [V]. The images and video sequences in the benchmark were captured over various urban/suburban areas of 14 different cities across China from north to south. Specifically, VisDrone2018 consists of 263 video clips and 10; 209 images (no overlap with video clips) with rich annotations, including object bounding boxes, object categories, occlusion, truncation ratios, etc. With intensive amount of effort, our benchmark has more than 2:5 million annotated instances in 179; 264 images/video frames [V]. Also Semantic Drone Dataset that focuses on semantic understanding of urban scenes for increasing the safety of autonomous drone flight and landing procedures. The imagery depicts more than 20 houses from nadir (bird's eye) view acquired at an altitude of 5 to 30 meters above ground. A high resolution camera was used to acquire images at a size of 6000x4000px (24Mpx). The training set contains 400 publicly available images and the test set is made up of 200 private images. [W].

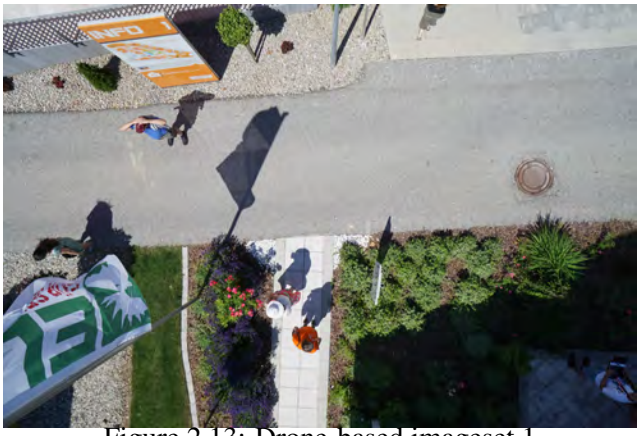

Figure 2.13: Drone-based imageset 1

Figure 2.14: Drone-based imageset 2

3. Chapter

Results and analysis

3.1 Used Technologies

Various technologies, frameworks and libraries were used in this project. Every implementation was carried out in Python, and frameworks like TensorFlow and Keras were used. Below are the list of the technologies used and a brief description.

3.1.1 Python

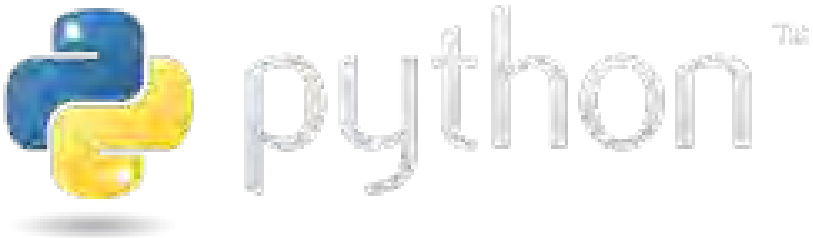

Figure 3.1: Python logo, source: https://www.python.org/community/logos/

Python[1] is a high level programming language designed by Guido van Rossum in 1991. Its design philosophy emphasizes code readability and has a remarkable use of significant whitespace. The filename extensions include .py, .pyc, .pyw, .pyz. It support web development with frameworks like Django and Pyramid. It is applied in scientific studies and machine learning. Several libraries and frameworks have been developed by the Python community for this such as SciPy[2], Scikit-learn[3], Pandas[4], Numpy[5], Theano[6], PyTorch [7], etc. It is also one of the most popular languages in the field of Convolutional neural network. Neural Networks like ResNet, VGGNet, Faster R-CNN, AlexNet in Keras or Pytorch, etc.

1. https://www.python.org/community/logos/
2. www.scipy.org
3. scikit-learn.org/stable/
4. pandas.pydata.org
5. www.numpy.org/
6. deeplearning.net/software/theano/
7. pytorch.org/

3.1.2 TensorFlow

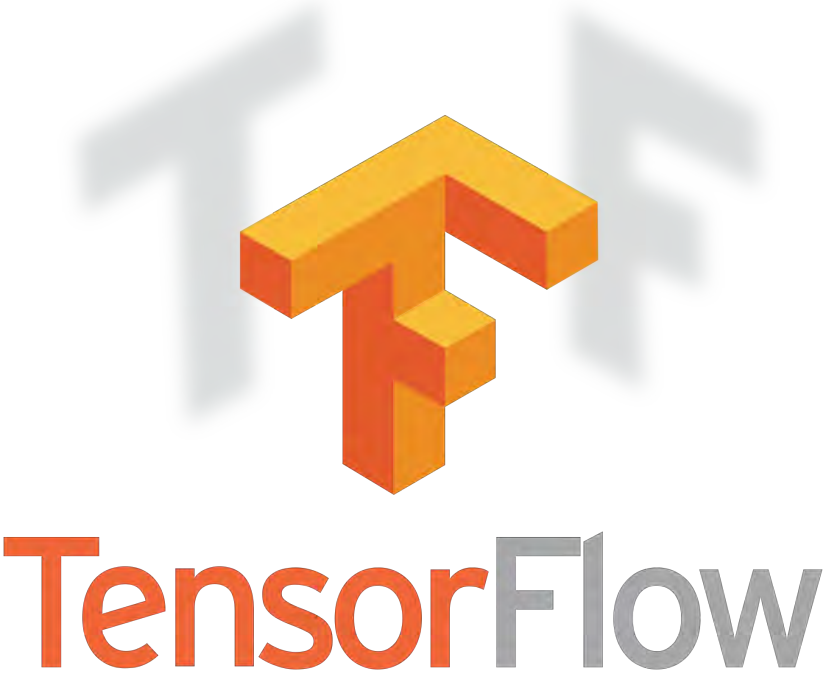

Figure 3.2: TensorFlow logo, source: www.tensorflow.org

TensorFlow[8] is an Apache Licensed library. It is free and open-sourced and developed by Google Brain Team. It is written in Python, C++ and CUDA. It is a symbolic math library applied for machine learning application such as deep neural network, convolutional neural network. TensorFlow was used in the implementation of the deep learning network for this project. TensorFlow has a comprehensive, flexible ecosystem of tools, libraries that makes it user-friendly. For a wider documentation, please see the official website[9]

8. www.tensorflow.org
9. www.tensorflow.org

3.1.3 Keras

Figure 3.3: Keras logo, source:www.scikit-image.org

Keras [10] is an MIT licensed neural-network library written in Python. It was design by Francois Chollet and released first in March 2015. It is open-sourced. Keras is capable of running on top of Microsoft Cognitive Toolkit, TensorFlow, PlaidML or Theano for fast experimentation of deep neural networks. It enables implementation of deep learning and allows easy and fast prototyping, supports both convolutional networks and recurrent networks and the combination of both. For a wider documentation, please see the official website[11]

3.1.4 Scikit-Image

Figure 3.4: Scikit-Image logo, source: www.scikit-image.org

Scikit-Image is a BSD licensed image processing library. It is a collection of algorithms for segmentation, geometric transformations, analysis, filtering, colour space manipulation, morphology, feature detection, and so on. It was first released in 2009, written by Stefan van der Walt.

10Keras.io
11www.keras.io

3.2 Implementation

3.2.1 Dataset

Drones, or general UAVs, equipped with cameras have been fast applied to a wide range of applications, including agricultural, aerial photography, fast delivery, and surveillance. Consequently, automatic understanding of visual data collected from these platforms become highly demanding, which brings computer vision to drones more and more closely. Various computer vision task has been carried out by the Vision meets Drone (VisDrone) Challenges[12] organized by the AISKEYE team at Lab of Machine Learning and Data Mining , Tianjin University, China in the year 2018. Various computer vision task carried out in the challenge include object detection in images. The task aims to detect objects of predefined categories (e.g., cars and pedestrians) from individual images taken from drones. Also, object detection in videos challenge. The task is similar to the first, except that objects are required to be detected from videos. Then single-object tracking challenge. The task aims to estimate the state of a target, indicated in the first frame, in the subsequent video frames. And finally, multi-object tracking challenge. The task aims to recover the trajectories of objects in each video frame. Originally, I wanted to use the dataset provided by VisDrone Challenge for the training and validation of my Mask R-CNN Network, but due to lack of the segmentation mask data for the training set, I couldn't use it. Rather I made use of the Semantic Drone Dataset from Institute of Computer Graphics and Vision, Graz University of Technology, Austria. [13]

The dataset contains images, bounding boxes as python pickle file, bounding boxes as xml, bounding boxes as mask images. The Semantic Drone Dataset focuses on semantic understanding of urban scenes for increasing the safety of autonomous drone flight and landing procedures. The imagery depicts more than 20 houses from nadir (bird's eye) view acquired at an altitude of 5 to 30 meters above ground. A high resolution camera was used to acquire images at a size of 6000x4000px (24Mpx). The training set contains 400 publicly available images and the test set is made up of 200 private images. [W] For the task of person detection the dataset contains bounding box annotations of the training and test set. And for semantic segmentation, pixel-accurate annotation for the same training and test set was prepared. The complexity of the dataset is limited to 20 classes as listed below.

- tree
- tree
- grass
- other vegetation
- dirt

12www.aiskyeye.com
13dronedataset.icg.tugraz.at

- gravel
- rocks
- water
- paved area
- pool
- person
- dog
- car
- bicycle
- roof
- wall
- fence
- fence-pole
- window
- door
- obstacle

The Drone Dataset is made freely available to academic and non-academic entities for non-commercial purposes such as academic research, teaching, scientific publications, or personal experimentation.

For the implementation of the instance segmentation, Python was used for obvious reasons. It enabled me to use deep learning framework like Tensorflow and Keras. The implementation of the model was carried out in the file drone.py and uses the Mask R-CNN library developed in python with TensorFlow and Keras by Matterport Inc. It was written by Waleed Abdulla from Matterport Inc. Matterport, Inc. published their implementation under the MIT License [X]. The MIT License is a license granting the permission to use the code, copy it, modify it, publish and even to sell it free of charge. There's absolutely no restriction on commercial use in MIT license. The only requirement is that you must include the MIT copyright notice with any copies of the software. Scripts, files and noteboks in the library are also under the MIT License and moreover, Waleed Abdulla himself agreed with the usage and modifications of his code for purposes of the research work.The Matterport, Inc. Mask R-CNN implementation can be found in their GitHub repository. Moreover, the Matterport implementation of Mask R-CNN was

selected because of several reasons. Besides its license compatibility, it is quite robust and ready for modifications and adaptation for training in drone dataset leading to another implementation, so it saved thousands of lines of code. One of the major motivation behind its usage is that there is a plenty of people interested in this project, proposing their ideas and testing it. And these people are experienced in fields of computer vision and deep learning. Abdulla himself is responsive and active in answering people's questions and issues. He is open-minded when discussing other people improvement proposals. I found it very useful. The next section will discuss the Mask R-CNN library, the workflow and necessary module of the library. The structure, architecture and components of the Mask R-CNN have been discussed in Chapter 2, so explanation of the programs only will be given in the next section. The library will be also discussed altogether with notes on my modifications connected with this thesis to distinguish them from Abdulla's code.

3.2.2 Mask R-CNN library

The library which is hosted in Github has been forked 5445 times. It contains five important modules. Each module play an important role in the training and testing of model. Let's go through this modules on after the other below:

Config.py

This is the configuration module of the system. It includes hyper-parameters for tuning of the model and necessary setting. It will described in section 4.2.1SS

Model.py This is fundamental core of the model. It develops and builds up the model. It will described in section 4.2.2

Parallel_Model.py This module subclasses the standard Keras Model and adds multi-GPU support. It works by creating a copy of the model on each GPU and creates a parallelized computation. This file was not modify further and so will not explained in this thesis.

Utils.py This modules contains the utility classes and functions of the model. It will described in section 4.2.3

Visualize.py This contains function for display and visualization. It will described in section 4.2.4

The python files for the modules have good inner documentation, of which some of them and other functionalities will be discussed next.

Config.py

Config.py is the configuration settings for the model in the implementation by Matterport Inc.. It contains the hyper-parameters of the model. It has a classed called ModelConfig that houses these parameters. These parameter include;

- GPU Count
- Images per GPU

- Steps per Epoch
- Validation steps
- The choice of the backbone
- Backbone strides (The strides of each layer of the Feature Pyramid Network (FPN) pyramid discussed in section 2.2)
- Learning Rate
- RPN Train Anchors Per_ Image (How many anchors per image to use for RPN training)
- Mask Shape
- And so on.

I inherited ModelConfig class and then adapted the attributes to suit my hardware and the drone-based dataset in the drone.py module. The config.py also a contains a display function that displays the model's attribute.

Model.py

This is the main Mask R-CNN implementation. It is made use of some modules for its implementation which include, os, random, datetime, re, logging, collections, multiprocessing ,numpy,tensorflow, keras, keras.backend ,keras.layers ,keras.engine , ,keras.models. distutils.version. For the module to work it needs TensorFlow 1.3+ and Keras 2.0.8+ . Since this module builds the Mask R-CNN it has several classes and function performing various tasks. This classes and task the perform can be summed up thus;

- Initialization functions
- Building the ResNet backbone.
- Building the RPN.
- Building RoIAlign layers.
- Building head architectures.
- Building the complete Mask R-CNN model and putting everything together.
- Building detection layers.
- Defining loss functions.
- Data formatting

– Miscellaneous functions and utilities connected to the model, like batch normalization data formatting and generating (building up targets, loading ground truth masks) or bounding boxes normalization.

Some important functions in the modules include train, load_weight, build, etc. There was no modification made in this module for the training of model, apart from few modification to fix error that could be raised during the loading of masks. An explanation is given to the program written by Waleed Abdulla

RESNET

The backbone of the Mask R-CNN is the Residual Network (RESNET). For the building of the RESNET, the principal function that does this is the resnet_graph . The resnet_graph builds the ResNet graph and the architecture can be ResNet50 or ResNet101.The batch norm layers of the network can be freeze or trained by setting the train_bn to True or False, but the default is True. The workflow is outlined in pseudocode 4.1

Algorithm 1: Building the ResNet backbone architecture

```
layers = intended layers
layers .add( zero padding 3x3)
layers .add( convolution 7x7)
layers .add( batch normalization )
layers .add( ReLu )
layers .add( maximum pooling )
layers .add( convolutional block 64 x64x256 )
layers .add (2 identity blocks 64 x64x256 )
layers .add( convolutional block 128 x128x512 )
layers .add (3 identity blocks 128 x128x512 )
layers .add( convolutional block 256 x256x1024 )
if architecture == 'resnet50 ' then
    layers .add (5 identity blocks 256 x256x1024 )
else if architecture == 'resnet101 ' then
    layers .add (22 identity blocks 256 x256x1024 )
return layers
```

.

The genuine function does not return total layers, however, it returns them in stages C1, C2, C3, C4, C5 as can be seen in pseudocode 4.8, where this function is called build_resnet_backbone. Every one of these stages speaks to the condition of craftsmanship before each convolutional block expansion, which is the last layer before changing components of input or yields. It is significant for the FPN as was referenced in Chapter 2 what's more, outlined in the model structure in pseudocode 4.8. Functions identity

block and convolutional block are fundamentally the same as and both fabricates the bottleneck block. The main contrast is that the convolutional block function likewise actualizes a 1x1 convolution in the alternate route association as it is important to change the state of the contribution to the one utilized in the block. The remainder of their usage is pretty much the equivalent and is represented in pseudocode .2 (the convolution ought to be connected in the yield association step). It uses channels given to each call of the capacity in the ResNet pseudocode.

Algorithm 2: identity_block

```
1  original_input = original_input_tensor
2  block = intended block of layers
3  block .add ( convolution 1x1)
4  block .add ( batch normalization )
5  block .add ( ReLu )
6  block .add ( convolution 3x3)
7  block .add ( batch normalization )
8  block .add ( ReLu )
9  block .add ( convolution 1x1)
10 block .add ( batch normalization )
11 block . connect_outputs (block , original_input )
12 block .add ( ReLu )
13 return block
```

.

RPN

The RPN is worked by two functions, build_rpn_model and rpn_graph. Notwithstanding, these functions assemble just the model, for example, the sliding window and its behavior, anchors are created in utils.py as depicted in section 5.1.3. Indeed, even in this split approach, it pursues the thought from section 3.4.3. Contributions for the rpn_graph capacity are an element map, number of anchors per area and anchors stride and returns anchors class logits, probabilities and bounding boxes refinements. The work process of rpn_graph is represented in pseudocode 5.3. build_rpn_model makes a model which initially feed the rpn_graph work and at that point restores the previously mentioned qualities.

Algorithm 3: rpn_graph

```
feature_map = input_feature_map
logits_number_of_filters = 2 * number of anchors per location
bbox_number_of_filters = 4 * number of anchors per location
shared_layer = convolution 3x3 on feature_map
rpn_class_logits = convolution 1x1 on shared_layer with logits_number_of_filters
rpn_probabilities = softmax on rpn_class_logits
rpn_bbox_refinements = convolution 1x1 on shared_layer with bbox_number_of_filters
return rpn_class_logits , rpn_probabilities , rpn_bbox_refinements
```

.

A significant class for the RPN is the ProposalLayer class. It takes anchor probabilities, bounding box refinements and anchors themselves as inputs trim them to littler clumps while considering top anchors and applies refinements to the anchor boxes.

Algorithm 4: ProposalLayer

```
probs = anchor probabilities
deltas = anchor refinements
anchors = anchors
threshold = threshold for probabilities
top_anchors = names_of_anchors_with_top_probs (probs , how_many =min(6000 , len( probs
)))
probs_batch = batch_slice (probs , top_anchors )
deltas_batch = batch_slice (deltas , top_anchors )
anchors_batch = batch_slice ( anchors , top_anchors )
boxes = apply_refinements ( anchors_batch , deltas_batch )
proposals = [boxes , probs_batch ]
proposals . apply_threshold ( threshold )
return proposals
```

.

ROIAlign

As was described already in chapter 2, RoIAlign is more or less the RoIPooling algorithm without rounding. The implementation is briefly sketched below

Algorithm 5: RoIAlign

```
1  pool_shape = shape of regions
2  image_shape = shape of the image
3  boxes = list of RoIs
4  feature_maps = list of feature maps
5  h, w = compute_heights_and_widths_boxes ( boxes )
6  image_area = image_shape [0] * image_shape [1]
7  roi_level = minimum (5, 4 + log2 ( sqrt (h * w) / (224 / sqrt (image_area ))))
8  pooled = list ()
9  for level in range (2, 6) do
10     roi_level_i = 1 where roi_level == level , 0 elsewhere
11     level_boxes = gather (boxes , indices = roi_level_i )
12     pooled . append ( crop_and_resize ( original_image = feature_maps [level-2] , what_process
         = level_boxes , shape = pool_shape , method =' bilinear '))
13 end
14 pooled . rearrange_to_match_the_order ( boxes )
15 return pooled
```

.

It executes the RoIAlign algorithm on different dimensions of the feature pyramid furthermore, in its specifications of the condition, it pursues the thoughts behind identifications in [27] and furthermore applies the five-levels approach. The base picking at line 7 and the loop at line 9 the then pursues utilizing just layers two to five from chapter 5.1.2.

Head architectures

As can be found in figure 3.13 and was at that point portrayed in section 3.6.1, the head architecture is separated into two areas. The head architecture for bounding boxes what's more, class probabilities are dealt with by the fpn_classifier_graph function and the mask architecture by the build_fpn_mask_graph. fpn_classifier_graph takes as input RoIs, feature maps, pool size and a number of classes and returns classifier logits, probabilities and bounding boxes refinements. build_fpn_mask_graph takes a similar input yet returns just a rundown of masks.

Algorithm 6: fpn_classifier_graph

```
rois = given regions of interest in normalized coordinates
feature_maps = list of feature maps from layers P2 , P3 , P4 , P5
pool_size = height of feature maps to be generated from ROIpooling
num_classes = number of classes
layers = list of keras layers
layers .add( ROIAlign ( pool_size , input =[ rois , feature_maps ]))
layers .add( convolution pool_size X pool_size )
layers .add( batch_normalization )
layers .add( ReLU )
layers .add( convolution 1x1)
layers .add( batch_normalizataion )
layers .add( ReLU )
shared = squeeze_to_one_tensor ( output of layers )
class_logits = fully_connected_layer ( input =shared ,number_of_filters = num_classes )
probabilities = softmax ( class_logits )
bboxes = fully_connected_layer ( input =shared , number_of_filters =4 * num_classes )
return class_logits , probabilities , bboxes
```

.

Algorithm 7: build_fpn_mask_graph

```
rois = given regions of interest in normalized coordinates
feature_maps = list of feature maps from layers P2 , P3 , P4 , P5
pool_size = height of feature maps to be generated from ROIpooling
num_classes = number of classes
layers = list of keras layers
layers .add( ROIAlign ( pool_size , input =[ rois , feature_maps ]))
layers .add( convolution 3x3)
layers .add( batch_normalization )
layers .add( ReLU )
layers .add( convolution 3x3)
layers .add( batch_normalization )
layers .add( ReLU )
layers .add( convolution 3x3)
layers .add( batch_normalization )
layers .add( ReLU )
layers .add( convolution 3x3)
layers .add( batch_normalization )
layers .add( ReLU )
layers .add( deconvolution 2x2 with strides 2)
layers .add( convolution 1x1 with sigmoid as an activation function )
return layers
```

.

In the pseudocodes above, a ROIAlign object is added as the first one into layers. This object was sketched in pseudocode 5.5.

Mask R-CNN model

The focal point of the model.py record is the MaskRCNN class which contains techniques to manufacture the whole Mask R-CNN model by cobbling together various kinds of layers what's more, to utilize it for training or detection. The work process of the technique build is represented in pseudocode 5.8 and pursues the architecture portrayed in chapter 3.6.1. In the pseudocode, we can see that the head architecture contrasts a bit in the training and in the detection. It is expected to the way that we need loss values to be processed during the training, so we figure them from detected values and target values (values dependent on known focuses from the training dataset).

Algorithm 8: Mask R-CNN.build

```
C2 , C3 , C4 , C5 = build_resnet_backbone ()
P5 , P4 , P3 , P2 = build_top_down_fpn_layers (C2 , C3 , C4 , C5)
anchors = generate_anchors ()
rpn = build_rpn ()
rois = ProposalLayer (rpn , anchors )
if mode == 'training ' then
    ground_truth_values = values from the training dataset
    bbox , classes = fpn_classifier ( rois )
    target_detection = DetectionTargetLayer ( ground_truth_values )
    mask = fpn_mask ( rois from target_detection )
    loss = loss_functions ( target_detection , bbox , classes , mask )
    model = [bbox , classes , mask , loss ]
else
    bbox , classes = fpn_classifier ( rois )
    target_detection = DetectionLayer (bbox , classes )
    mask = fpn_mask ( rois )
    model = [bbox , classes , mask ]
end
return model
```

.

In the pseudocode, we can see a few classes and functions. Despite the fact that their motivations are very clear, some of them can be seen in various pseudocodes. Function build_resnet_backbone was at that point depicted in pseudocode 5.1, ensuing function build_top_down_fpn_layers is genuinely clear procedure connecting layers as in section 3.6.1, generate_anchors will be portrayed in 5.12, build_rpn can be seen in pseudocode 5.3, ProposalLayer in pseudocode 5.4, fpn_classifier speaks to the fpn_classifier_graph from pseudocode 5.6 and fpn_mask is work build_fpn_mask_graph from pseudocode 5.7.

utils.py

The most significant piece of the utils.py record is the Dataset class. It is likewise the main some portion of the utils.py code that was adjusted for the necessities of Drone-based dataset utilization (the other changes are simply minor refactorings). The utils.py additionally contains a great deal of functions. Just a couple of them will be referenced as every one of them have adequate documentation in the code.

Dataset

The Dataset class is the base class for dataset classes and images. It contains data about them including

their names, identifiers and on account of images

likewise paths to them. One of the written methods is the one called import_contents, which feeds the Dataset object with classes and images. The work process is represented in pseudocode

5.9. Contributions for the technique are:

- List of classes names proposed to be learned
- List of directories containing training images and masks
- Name of model

The add_class method in pseudocode 5.9 import a class into the Dataset object dictionary inside and out with an exceptional identifier; a significant part is containing the foundation as the first class with identifier 0 (in the pseudocode represented simplifiedly by the saved_class dictionary). The add_images line is a loop over all images with the predefined augmentation contained in a given registry bringing in them out and out with their identifier and way into the Dataset object list.

Algorithm 9: import_contents

```
1 classes = list of classes names intended to be learned
2 directories = list of directories containing training images and masks
3 saved_classes = 'BG ': 0
4 for i in classes do
5 |   add_class
6 end
7 for directory in directories do
8 |   add_images
9 end
```

.

Another important method written for the needs of the Drone-based dataset modules is the one called load_mask. The work process of the method is illustrated in pseudocode 5.10. It returns an array containing boolean masks (True for the mask, False elsewhere) for each instance in the image, an array of class identifiers corresponding each instance in the masks array.

Algorithm 10: get_mask

```
masks_list = list of mask files within the directory
first_mask = masks_list [0]
masks_array = array containing first_mask transformed to bool
classes_list = list containing class of the first mask
for new_mask in masks_list [1:] do
    concat_mask = new_mask transformed to bool
    concatenate masks_array with concat_mask
    append class of new_mask into classes_list
    if any problem happened then
        return None , None
end
return masks_array , classes_list
```

.

From the remainder of Dataset class methods, one more will be referenced. prepare must be called before the use of the Dataset object as it sets it up for use. The readiness is done through setting object parameters like number of classes, classes names and identifiers or number of pictures. This setting depends on data got during the import_contents call. Bounding boxes apparatuses Since bounding boxes are not required to be given altogether with masks in the training dataset, the function extract_boxes is utilized to process bounding boxes from masks. The function scans for the first and last horizontal and vertical positions containing mask along all channels and returns them as an array. It implies that every pixel of the mask is contained in the returned horizontal-vertical bounding box and it is likewise as tight as could reasonably be expected. A function used to register the IOU is called just compute_iou. Its work process is delineated in pseudocode 5.11. The treatment of no interception is likewise executed in the function, yet for better understanding, it is excluded in the pseudocode.

Algorithm 11: compute_iou

```
predicted_box_area = area of predicted box
groundtruth_box_area = area of given mask
y1 = the bigger one from the upper coordinates of the predicted and ground truth bboxes
y2 = the smaller one from the lower coordinates of the predicted and ground truth bboxes
x1 = the bigger one from the left coordinates of the predicted and ground truth bboxes
x2 = the smaller one from the right coordinates of the predicted and ground truth bboxes
intersection = (x2 - x1) * (y2 - y1)
union = predicted_box_area + groundtruth_box_area - intersection
iou = intersection / union
return iou
```

.

With the correlation of ground truth boxes and the anticipated ones is associated likewise the function box_refinement. It registers contrasts between ground truth furthermore, anticipated directions of bounding boxes and returned them as the data of the error bounding box incorrectness.

Pyramid anchor tools

The hypothesis of scales and pyramids was at that point portrayed in chapter 3.4.3 and 3.6.1. Two functions are associated with the formation of the anchors at various dimensions of a feature pyramid. The called one is generate_pyramid_anchors which loops over scales. Tuned in, the generate_anchors capacity is called to produce anchors of proportions for a given arrangement of scales. The work process of the generate_anchors function is represented in pseudocode 5.12. It takes scales and proportions of anchors, feature map shape and anchors and feature map strides as a series of inputs. It utilizes these contributions to figure heights and widths of various stays (can be found in figure 3.6) and to compute a framework of anchors focuses. This network together with their height and widths characterizes the returned value, anchors.

Algorithm 12: generate_anchors

```
scales = array of scales
ratios = array of ratios
feature_map_shape = [height , width ]
anchor_stride = stride of anchors on the featuremap
feature_stride = stride of the featuremap
heights = scales divided by a square root of ratios ( each by each )
widths = scales multiplied by square root of ratios ( each by each )
shifts_y = grid from 0 to shape [0] with stride anchor_stride
shifts_y = shifts_y * feature_stride
shifts_x = grid from 0 to shape [1] with stride anchor_stride
shifts_x = shifts_x * feature_stride
anchors_centers = stack of [ shifts_y , shifts_x ] in each combination
anchors_sizes = [ heights , widths ]
anchors = [ anchors_centers - 0.5 * anchors_sizes , anchors_centers + 0.5 * anchors_sizes ]
return anchors
```

.

Drone.py The responsibility of parents in teaching and training their children. The responsibility of drone.py is to train the model to detect or predict object. Drone.py teaches the model to recognize the object attributes, colour, curves, edges and other attributes of object. This file was written as part of the practical section of the thesis and defines an important function- train. The train function in drone.py calls the parent train function present in the Mask R-CNN model defined above. Before the training can

take place the dataset needs to be loaded and likewise the mask. So the call to the load_drone function loads the drone-bades dataset and their corresponding mask. After the load_drone function the next is the call to the prepare function. The prepare function sets up the model for training. The flowchart contains couple of as of now referenced functions and classes, explicitly the ModelConfig class and its display method from chapter 5.1.1, the MaskRCNN class from section 5.1.2 and the Dataset class and its methods import_contents what's more, prepare from chapter 5.1.3. The last advance in this flowchart, a method model.train(), has really two extraordinary structures relying upon the utilization of initial weights. The initial structure is connected for a training from a scratch and prepares all layers. The second one comprises of three littler sections; right off the bat preparing layers 5 and higher, at that point adjusting layers 4 and higher and the last and greatest section is tweaking the entire design. It is appeared in the flowchart in 15 and the thought behind this conduct is that it is unrealistic to train the first layers including low-level features, while changes have a tremendous effect on more profound dimensions and those highlights ought to be pretty much the equivalent for any object.

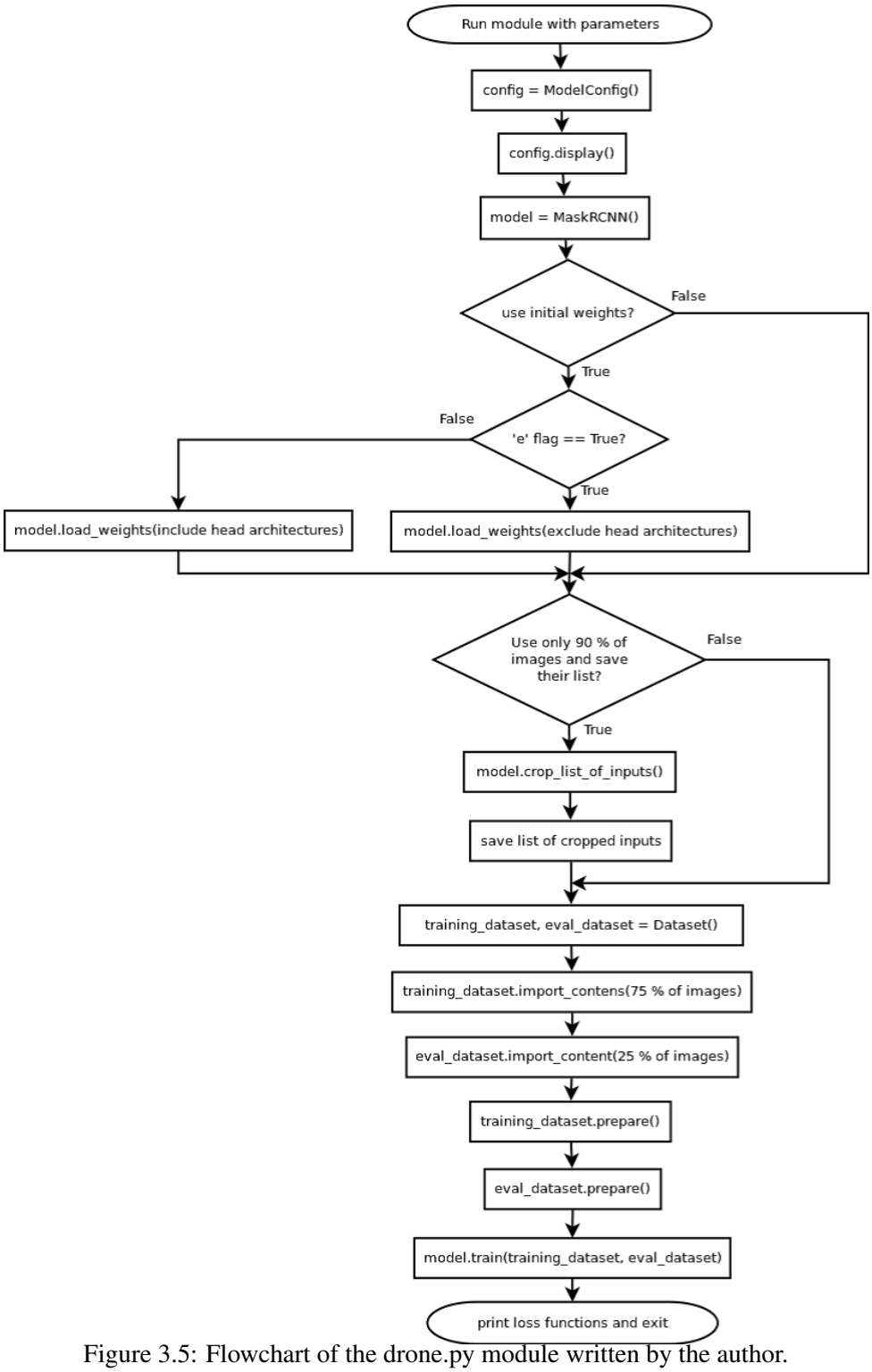

Figure 3.5: Flowchart of the drone.py module written by the author.

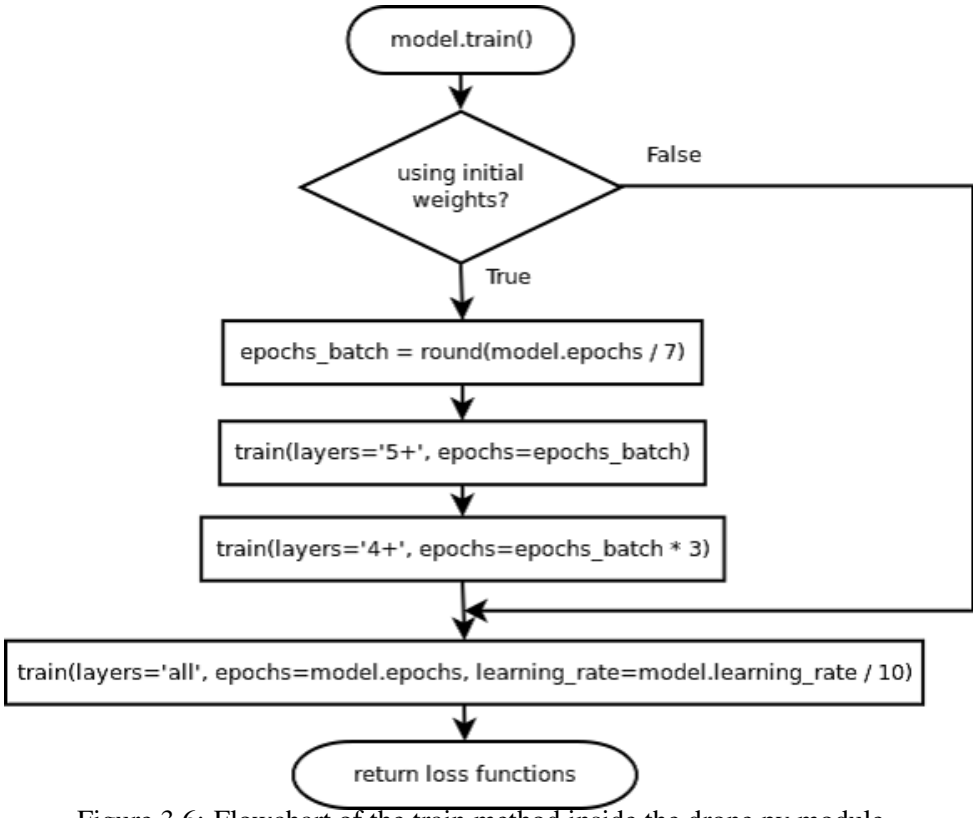

Figure 3.6: Flowchart of the train method inside the drone.py module .

Drone-detect.py In the event that we have a trained model, either prepared by us or given by another person, we can utilize it to distinguish highlights or items in maps. In fact, the trained model is used to detect features and maps in drone datasets collected from drones developed by the Robotic Team of African University of Science and Technology. The yield from the module comprises of a set of vector maps for each class. Despite the fact that the model is somewhat scale-invariant, it is prescribed to give object detection and masks incomparable goals to the one utilized in training pictures. The module is contained in the document drone_detect.py. This record can be viewed as the second piece of the down to earth segment of the theory. Its work process is with certain improvements. As in the drone.py module, the flowchart contains few already mentioned functions and classes, explicitly the ModelConfig class and the MaskRCNN class. Be that as it may, more unmentioned function can be found in the detection module.

4. Chapter

Conclusion

Summary: *The aim of this project is to automatically detect and segment objects in a drone-based dataset by extending the application of Mask R-CNN the drone-based dataset. With the advance of the applicability of drone to daily-life activities, industries and farms. Counting objects with drones is a new application of drone. This work aim to provide a system for object detection in drone-based dataset that can be practically applied in facilitating work performed by industries like agricultural, oil and gas industries and so on. As a future work the model will be deployed in the drone developed by the Robotic Team of African University of Science and Technology (AUST) for detection and control of pipeline vandalism in oil and gas industries. The first part of the thesis was dedicated to a theoretical background behind CNNs.It also discussed by the general overview of various computer vision tasks. The second part is dedicated to the introduction of tools used in the work and the implementation of Mask R-CNN modules using the technologies on drone-based datasets. More importantly explanation to most important parts of the code. Developed modules are available in GitHub repository.*

4.1 Programming summary

A. Appendix

Principal program codes

We see the program in the **drone.py module** , see [Heinz and Moses, 2007], that is well written and documented.

```
# -*- coding: utf-8 -*-

import os
import sys
import datetime
import numpy as np
import skimage.draw
import pandas as pd
import tensorflow as tf


# Root directory of the project
ROOT_DIR = os.path.abspath("C:\\Users\\HP USER\\Desktop\\francis\\Mask_RCNN")
#ROOT_DIR = "Mask_RCNN/"

# Import Mask RCNN
sys.path.append(ROOT_DIR)  # To find local version of the library
from mrcnn.config import Config
from mrcnn import model as modellib, utils

# Getting neccessary directories
dir_name= os.fspath('Dataset-Cut/train')
image_dir=os.path.join(dir_name,'train_images')
mask_dir=os.path.join(dir_name,'mask')
class_rgb=os.path.join(dir_name,'class_dict.csv')

# Path to trained weights file
COCO_WEIGHTS_PATH = os.path.join(dir_name, "mask_rcnn_coco.h5")

# Directory to save logs and model checkpoints, if not provided
# through the command line argument --logs
DEFAULT_LOGS_DIR = os.path.join(dir_name)


#Reading the files
class_rgb = pd.read_csv(class_rgb,delimiter=',') #The class and RGB values for the masks in
    the final output

"""CONFIGURATION"""

class DroneConfig(Config):
    """Configuration for training on the drone  dataset.
    Derives from the base Config class and overrides some values.
    """

    NAME = "drone"

    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 22  # Background + others

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 100

    # Skip detections with < 90% confidence
    DETECTION_MIN_CONFIDENCE = 0.9

class DroneInferenceConfig(DroneConfig):
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    RPN_NMS_THRESHOLD = 0.7
```

A. Appendix: Principal program codes

```
# Tryig to create the imageids from the xml files, not adviced. (SKIP)
def read_xml():
    parser = ET.XMLParser(encoding="iso-8859-5") # Parser for XML
    #The XML file handling
    tree = ET.parse(tree)
    root = tree.getroot()
    child_tag=[]
    image_ids=[]
    for child in root:
        child_tag.append((child.tag, child.attrib))
    elem_tag=[elem.tag for elem in root.iter()]
    xml_string=ET.tostring(root, encoding='utf8').decode('utf8')
    #print(xml_string)
    #SI_attrib=[sourceImage.attrib for sourceImage in root.iter("sourceImage")]
    #SI_text=[sourceImage.text for sourceImage in root.iter("sourceImage")]
    #Fname_text=[filename.text for filename in root.iter("filename")]
    for item in ["%03d" % i for i in range(1,599)]: # Creating 3-digits figures to suit the
        xml file naming
        try:
            tree= os.path.join(annotations,str(item)+'.xml')
            tree = ET.parse(tree)
            root = tree.getroot()
            file_name_text=[filename.text for filename in root.iter("filename")]
            file_name_text=", ".join(map(str, file_name_text))  #Removing the bracket before
                appending to image_ids
            image_ids.append(file_name_text)
        except (FileNotFoundError,ParseError):
            pass

"""DATASET"""

class DroneDataset(utils.Dataset):
  def load_drone(self, dataset_dir, subset):
      class_name=class_rgb['name'].tolist()
      for i,classes in enumerate(class_name[1:23],1):
          self.add_class("drone", i, "drone")
          #Which subset?
      #assert subset in ["train", "val"]
      #train="/train/train_images"
      #val="/test/test_images"
      #subset_dir = train if subset in ["train"] else val
      # Get image ids from directory names
      #dataset_dir = os.path.join(dataset_dir, subset_dir)

      #image_ids = next(os.walk(dataset_dir))[1]
      #image_ids= list(set(image_ids))

      image_ids=['013.jpg',
 '028.jpg',
 '008.jpg',
 '022.jpg',
 '019.jpg',
 '003.jpg',
 '026.jpg',
 '040.jpg',
 '041.jpg',
 '001.jpg',
 '005.jpg',
 '016.jpg',
 '004.jpg',
 '014.jpg',
 '021.jpg',
 '023.jpg',
 '011.jpg',
 '006.jpg',
 '015.jpg',
 '018.jpg',
 '038.jpg',
 '035.jpg',
 '002.jpg',
 '031.jpg']
      # Add images
      for image_id in image_ids:
          self.add_image(
                  "drone",
                  image_id=image_id,
                  path=os.path.join(dataset_dir, "train", "train_images/{}".format(image_id)))
      #try:
      #except StopIteration as e:
      #  pass

      def load_mask(self, image_id):
          info = self.image_info[image_id]
          # Get mask directory from image path
          mask_dir = os.path.join(os.path.dirname(os.path.dirname(info['path'])), "masks")
          # Create a cache directory
          # Masks are in multiple png files, which is slow to load. So cache
```

A. Appendix: Principal program codes

```
            # them in a .npy file after the first load
            cache_dir = os.path.join(dir_name, "/cache")
            if not os.path.exists(cache_dir):
                os.makedirs(cache_dir)
            # Is there a cached .npy file?
            cache_path = os.path.join(cache_dir, "{}.npy".format(info["id"]))
            if os.path.exists(cache_path):
                mask = np.load(cache_path)
            else:
                # Read mask files from .png image
                mask = []
                for f in next(os.walk(mask_dir))[2]:
                    if f.endswith(".png"):
                        m = skimage.io.imread(os.path.join(mask_dir, f)).astype(np.bool)
                        mask.append(m)
                mask = np.stack(mask, axis=-1)
                # Cache the mask in a Numpy file
                np.save(cache_path, mask)
            # Return mask, and array of class IDs of each instance.
            a= list(range(1,22))
            return mask,np.array(a,dtype=np.int32)

def train(model, dataset_dir, subset):
    """Train the model."""
    # Training dataset.
    dataset_train = DroneDataset()
    dataset_train.load_drone(dataset_dir, subset)
    dataset_train.prepare()

    # Validation dataset
    dataset_val = DroneDataset()
    dataset_val.load_drone(dataset_dir, "val")
    dataset_val.prepare()

    #Training the head layer i.e. the early layers of the network
    print("Train network heads")
    model.train(dataset_train, dataset_val,
                learning_rate=config.LEARNING_RATE,
                epochs=2,
                layers='heads')

    # Training all the network
    print("Train all layers")
    model.train(dataset_train, dataset_val,
                learning_rate=config.LEARNING_RATE,
                epochs=2,
                layers='all')

"""ENCODING"""

"""
Since our mask are images in png file format we need to encode them so that we can be able to
    use them.
We will do using the Run Length Encoding (RLE).Run-length encoding (RLE) is a simple form of
    data compression,
where runs (consecutive data elements) are replaced by just one data value and count.
"""
def rle_encode(mask):
    """Encodes a mask in Run Length Encoding (RLE).
    Returns a string of space-separated values.
    """
    assert mask.ndim == 2, "Mask must be of shape [Height, Width]"
    # Flatten it column wise
    m = mask.T.flatten()
    # Compute gradient. Equals 1 or -1 at transition points
    g = np.diff(np.concatenate([[0], m, [0]]), n=1)
    # 1-based indicies of transition points (where gradient != 0)
    rle = np.where(g != 0)[0].reshape([-1, 2]) + 1
    # Convert second index in each pair to lenth
    rle[:, 1] = rle[:, 1] - rle[:, 0]
    return " ".join(map(str, rle.flatten()))


def rle_decode(rle, shape):
    """Decodes an RLE encoded list of space separated
    numbers and returns a binary mask."""
    rle = list(map(int, rle.split()))
    rle = np.array(rle, dtype=np.int32).reshape([-1, 2])
    rle[:, 1] += rle[:, 0]
    rle -= 1
    mask = np.zeros([shape[0] * shape[1]], np.bool)
    for s, e in rle:
        assert 0 <= s < mask.shape[0]
        assert 1 <= e <= mask.shape[0], "shape: {}  s {}  e {}".format(shape, s, e)
        mask[s:e] = 1
    # Reshape and transpose
    mask = mask.reshape([shape[1], shape[0]]).T
```

A. Appendix: Principal program codes

```
    return mask


def mask_to_rle(image_id, mask, scores):
    "Encodes instance masks to submission format."
    assert mask.ndim == 3, "Mask must be [H, W, count]"
    # If mask is empty, return line with image ID only
    if mask.shape[-1] == 0:
        return "{},".format(image_id)
    # Remove mask overlaps
    # Multiply each instance mask by its score order
    # then take the maximum across the last dimension
    order = np.argsort(scores)[::-1] + 1  # 1-based descending
    mask = np.max(mask * np.reshape(order, [1, 1, -1]), -1)
    # Loop over instance masks
    lines = []
    for o in order:
        m = np.where(mask == o, 1, 0)
        # Skip if empty
        if m.sum() == 0.0:
            continue
        rle = rle_encode(m)
        lines.append("{}, {}".format(image_id, rle))
    return "\n".join(lines)

"""DETECTION"""

def detect(model, dataset_dir, subset):
    """Run detection on images in the given directory."""
    print("Running on {}".format(dataset_dir))

    # Create directory
    if not os.path.exists(RESULTS_DIR):
        os.makedirs(RESULTS_DIR)
    submit_dir = "submit_{:%Y%m%dT%H%M%S}".format(datetime.datetime.now())
    submit_dir = os.path.join(RESULTS_DIR, submit_dir)
    os.makedirs(submit_dir)

    # Read dataset
    dataset = DroneDataset()
    dataset.load_drone(dataset_dir, subset) #ATTENTION
    dataset.prepare()
    # Load over images
    submission = []
    for image_id in dataset.image_ids:
        # Load image and run detection
        image = dataset.load_image(image_id)
        # Detect objects
        r = model.detect([image], verbose=0)[0]
        # Encode image to RLE. Returns a string of multiple lines
        source_id = dataset.image_info[image_id]["id"]
        rle = mask_to_rle(source_id, r["masks"], r["scores"])
        submission.append(rle)
        # Save image with masks
        visualize.display_instances(
            image, r['rois'], r['masks'], r['class_ids'],
            dataset.class_names, r['scores'],
            show_bbox=False, show_mask=False,
            title="Predictions")
        plt.savefig("{}/{}.png".format(submit_dir, dataset.image_info[image_id]["id"]))

    # Save to csv file
    submission = "ImageId,EncodedPixels\n" + "\n".join(submission)
    file_path = os.path.join(submit_dir, "submit.csv")
    with open(file_path, "w") as f:
        f.write(submission)
    print("Saved to ", submit_dir)

"""COMMAND LINE"""

if __name__ == '__main__':
    import argparse

    # Parse command line arguments
    parser = argparse.ArgumentParser(
        description='Mask R-CNN for drone counting and segmentation')
    parser.add_argument("command",
                        metavar="<command>",
                        help="'train' or 'detect'")
    parser.add_argument('--dataset', required=False,
                        metavar="image_dir",
                        help='Root directory of the dataset')
    parser.add_argument('--weights', required=False,
                        default=os.path.join(dir_name,'/mask_rcnn_coco.h5'),
                        metavar="COCO_WEIGHTS_PATH",
                        help="Path to weights .h5 file or 'coco'")
    parser.add_argument('--logs', required=False,
```

A. Appendix: Principal program codes

```
                        default=DEFAULT_LOGS_DIR,
                        metavar=os.path.join(dir_name,'/logs'),
                        help='Logs and checkpoints directory (default=logs/)')
    parser.add_argument('--subset', required=False,
                        metavar="Dataset sub-directory", #ATTENTION
                        help="Subset of dataset to run prediction on")
    args = parser.parse_args()

    # Validate arguments
    if args.command == "train":
        assert args.dataset, "Argument --dataset is required for training"
    elif args.command == "detect":
        assert args.subset, "Provide --subset to run prediction on"

    print("Weights: ", args.weights)
    print("Dataset: ", args.dataset)
    if args.subset:
        print("Subset: ", args.subset)
    print("Logs: ", args.logs)

    # Configurations
    if args.command == "train":
        config = DroneConfig()
    else:
        config = DroneInferenceConfig()
    config.display()

    # Create model
    if args.command == "train":
        model = modellib.MaskRCNN(mode="training", config=config,
                                  model_dir=args.logs)
    else:
        model = modellib.MaskRCNN(mode="inference", config=config,
                                  model_dir=args.logs)

    # Select weights file to load
    if args.weights.lower() == "coco":
        weights_path = COCO_WEIGHTS_PATH
        # Download weights file
        if not os.path.exists(weights_path):
            utils.download_trained_weights(weights_path)
    elif args.weights.lower() == "last":
        # Find last trained weights
        weights_path = model.find_last()
    elif args.weights.lower() == "imagenet":
        # Start from ImageNet trained weights
        weights_path = model.get_imagenet_weights()
    else:
        weights_path = args.weights

    # Load weights
    print("Loading weights ", weights_path)
    if args.weights.lower() == "coco":
        # Exclude the last layers because they require a matching
        # number of classes
        model.load_weights(weights_path, by_name=True, exclude=[
            "mrcnn_class_logits", "mrcnn_bbox_fc",
            "mrcnn_bbox", "mrcnn_mask"])
    else:
        model.load_weights(weights_path, by_name=True)

    # Train or evaluate
    if args.command == "train":
        train(model, args.dataset, args.subset)
    elif args.command == "detect":
        detect(model, args.dataset, args.subset)
    else:
        print("'{}' is not recognized. "
              "Use 'train' or 'detect'".format(args.command))

"""FOR TRAINING IN PYTHON NOTEBOOK only"""
def notebook_train():
    DEVICE = "/gpu:0"  # /cpu:0 or /gpu:0
    MODEL_DIR = os.path.join(dir_name, "logs")

    config = DroneConfig()

    with tf.device(DEVICE):
        model = modellib.MaskRCNN(mode="training", model_dir=MODEL_DIR,
                                  config=config)
    weights_path = "Dataset-Cut/train/mask_rcnn_coco.h5"

    print("Loading weights ", weights_path)
    model.load_weights(weights_path, exclude=[
                "mrcnn_class_logits", "mrcnn_bbox_fc",
                "mrcnn_bbox", "mrcnn_mask"])
```

```
406     path='Dataset-Cut'
        train(model, path, 'train')
```

Bibliography

C. Heinz and B. Moses. The listings package. Technical report, CTAN TEX Archive, 2007. URL http://www.ctan.org/tex-archive/macros/latex/contrib/listings.