Artificial intelligence based automatic meter reading framework

A final year project report submitted to
COMSATS University Islamabad, Sahiwal
in the fulfilment of the requirements for the degree of
Bachelor of Science in Electrical Engineering
by

Saad Masrur	FA17-BEE-037
Aoun Abbas	FA17-BEE-063
Muhammad Kashif Riaz	FA17-BEE-080

Supervisor:
Dr. Saqib Saleem

Co-Supervisor:
Mr. Hamid Saeed

Department of Electrical & Computer Engineering
COMSATS University Islamabad, Sahiwal Campus

June 2021

Abstract

Approximately 28 million consumers are being facilitated by 10 power distribution companies of all Pakistan (excluding the area served by K-Electric). Water and Power Development Authority (WAPDA), being a central entity, is responsible for the current billing system mainly relying on the manual meter reading process, whereby a meter reader notes down readings on a register and takes an image of the meter as a proof-of-reading. Subsequently, the written reading is compared with the meter-image for verification purposes, followed by the issuance of a consumption bill. This whole process is repeated every month for each meter, thus requiring a lot of human effort and time. In line with the Punjab Information Technology Company (PITC)'s initiative of mobile meter reading the system, this project aims to develop and design an artificial intelligence based framework which could assist humans with the automatic meter reading. With the purpose to reduce wrong meter reading complaints and reducing billing cycle time, the key objective of this research is to replace the manual meter reading process with an automatic meter reading protocol. To achieve this, an image-based automatic meter reading scheme will be developed employing state-of-the-art deep machine learning architectures.

Acknowledgements

First of all, all laudable praises to Allah Almighty, who illuminates our path and leads us to decency and decency. This project would not have been completed without the continued help and support of several people whom I would like to mention and thank.

We are extremely grateful to our esteemed leader Dr. Saqib Saleem, Head of Electrical and Computer Engineering Department for their valuable advice, guidance, helpful discussions, and support throughout our research. Apart from valuable academic advice and guidance, he was extremely kind, friendly, and helpful.

Special thanks to Hamid Saeed, for helping us in multiple ways throughout this research project.

We also want to thank the Project Appraisal Committee (PAC) for its valuable advice and guidance.

Special thanks to our parents, brothers, sisters, and friends for their support, patience, and love. Without their support, motivation, and understanding, we would not have been able to complete this work. Finally, we sincerely thank all the people who supported us in completing this work.

Contents

List of Figures

List of Tables

Abbreviations

PITC	**P**unjab Information Technology **C**ompany
WAPDA	**W**ater and Power **D**evelopment Authority
AMR	**A**utomatic Meter **R**eading
OCR	**O**ptical Character **R**cognition
MLP	**M**ulti Layer Perceptron
CCA	**C**onnected Component Analysis
SVM	**S**upport Vector Machine
VEDA	**V**ertical Eedge Detection Algorithm
LSTM	**L**ong Short Term Memory
CNN	**C**onvolutional Neural Network
YOLO	**Y**ou Only Look Once
RCNN	**R**ecurrent Convolutional Neural Network
F-RCNN	**F**aster Recurrent Convolutional Neural Network
CCD	**C**omsats Counter Detection
CDRD	**C**omsats Digit Recognition Detection

Chapter 1

Introduction

Automatically monitoring the usage of electricity, water, and gas for billing and checking is known as Automatic Meter Reading (AMR) [1]. Although many developed countries have replaced the manual meter reading with the automatic process [2], but still in underdeveloped countries meter reading is executed manually [3]. Where a meter reader takes a picture of the meter, notes down the electricity consumption from the image, and places the image on the bill as proof. Overall, the efficiency of this meter reading procedure is quite low as it requires a lot of human effort and time moreover this process is prone to error due to human involvement at various stages. In this process of manual meter reading, a large number of images are needed to estimate to record the energy consumption, this whole process of evaluation is done by sampling, so there are very high chances for error to occur [4]. The problems associated with manual meter reading can be curbed by incorporating an automatic meter reading scheme. The error due to human factors can be effectively reduced by its introduction further human resources can be saved to a larger extent. Moreover, placing the cameras inside the meter can lead to automatic meter reading but this process will require more financial resources [5]. Installation of new meters is not required in the Image-based AMR approach which is a more cheap and effective solution when resources are scarce as no installation of a new meter is required. .

1.1 Problem Statement

The electricity consumers in Pakistan are mostly unsatisfied due to over-billing complaints. This is largely because meter readers do not go for actual meter reading and report either average consumption of same month from the last year on their registers or report extra consumption's to minimize aggregate technical commercial losses. This project will address the problem of these inaccurate meter readings which is due to the human made errors by meter readers. Due to errors in meter reading, the consumers suffer, and they have to take necessary steps to get the bill corrected from the concerned authorities. This issue also affects the utility companies in the form of payment delays and additional processing. To this, the proposed project will develop a framework which will effectively eliminate errors in the meter reading process. Specifically, we will use artificial intelligence (AI) based framework to obtain the correct information from the image of a meter taken by the meter reader's mobile phone [1]. In this way, the meter reader does not have to manually update the billing details, in fact, the consumer bills will be generated automatically from the images taken by the meter reader. The proposed pipeline will reduce the probability of error in the billing process significantly, as the bill will be generated automatically from the image of the meter. To sum, a meter reader will take snap of a meter and a bill will be automatically generated without any human intervention.

1.2 Scope and Research Objectives

With the purpose to reduce wrong meter reading complaints and reducing billing cycle time, the key objective of this research is to replace the manual meter reading process with an automatic meter reading protocol. To achieve this, an image-based automatic meter reading scheme will be developed employing advanced deep learning techniques. The main beneficiary of the proposed project will be Pakistan's power sector, to be specific, with its use there will be: .

- No need to manually note down meter reading by meter readers,
- Error of wrong meter reading will be potentially reduced,
- Intentional fraud associated with a meter reader would be diminished,
- Manual work of data entry to prepare monthly bills (and comparing with meter image) will be replaced by an automatized process.

Of note, the scope of the developed framework can be extended to other meter reading applications including water and gas meters. To achieve the above-mentioned objectives, the aims of the proposed project are: .

- Develop a framework requiring minimal involvement of humans for its use,
- develop a robust protocol which is least sensitive to environmental considerations, and
- user-friendly to staff.

1.3 Activities to be taken to Achieve Project Goals

The proposed scheme will be developed by performing following activities: .

- Develop a large database of raw images of meters. Knowing deep learning architectures require big data for their training and testing phases, we will aim to collect 10k+ images of all (accessible) types of meters installed in Pakistan,
- Comprehend functional basis of modern deep learning architectures which have been developed and adopted for similar deep learning tasks e.g., object localization and object recognition
- Construct a novel deep learning framework which will be trained and optimized on our database providing an acceptable accuracy,
- Develop an easy-to-use software routine which will be installed on a circle-level central computer for the real time evaluation of the proposed framework.

1.4 Output's need or Relationship to Industry

Currently, 23k+ meter readers are working with WAPDA for manual meter reading, where they are also responsible to record pictures of meters being printed on bills as proof-of-reading. Despite sincere efforts, this meter reading strategy has been found error-prone, mainly attributed to human involvement. To this, PITC, as an IT solution provider to DISCOs, has taken an initiative to develop a mobile meter reading system. As an extension and improvement to this existing reading system, the key proposal is to develop an automatic billing system by adding

artificial intelligence where human dependency is minimum. For this, we have decided to develop an image-based framework that will be easy to use by meter reading staff and will involve minimum human factors.

Chapter 2

Literature Review

License plate [6] and robust reading [7] which are optical character recognition (OCR) used to get information from an image which somehow resembles AMR. These techniques are widely spread in the literature, but AMR is not much discussed. Here is a brief survey of previous work.

2.1 Optical Character Recognition (OCR)

Recognition of license plate and robust reading are the applications of OCR which are based on extraction of text (information) from the pictures concerning challenging circumstances and special conditions. Although, work regarding Automatic Meter Reading also known as AMR is not presented by literature [8], as of applications OCR, recently work worth satisfying is produced [9].

For automatic meter reading (AMR), the initial errors of rotating digits produced in the system are because of unnecessary challenges in OCR. For recognition of digits, the methods regarding robust reading are already used but the problem is still there and many of the other methods had been proposed for the solution of this problem. With the use of a method based on Hausdorff distance, the problem is also presented by Rodriguez and Berdug et al.[10].

2.1.1 Limitations

In real time, this method gained results worth well recognition and by using a single meter, all pictures extraction was completed. The requirement of controlled environment was necessary because for the pre-processing and correction of angle, no method was used at all.

2.2 Multi-layer Perceptron (MLP)

For the segmentation of digits and detection of the counter, MLP also known as Multi-layer Perceptron was used by Nodari and Gallo et al.[3] without doing pre and post-processing. For the neglection of false positives and improvement in the detection of the counter, they added some methods [10] because they obtained low F-measures. They obtained superb results because of the use of methods of Fourier Analysis and watershed.

2.2.1 Limitations

Only 100 images were used to check the accuracy of the of the applied technique, which is not enough.For the firs time images were made publicly available for other experiments.

2.3 Vertical and Horizontal Pixel Projection Histogram

For the detection of the counter, a lot of approaches dealt with the horizontally and vertically projection of pixels in the histogram. The methods proposed in [1,8] which are based on projections are going to affect very easily along with the counter-rotation. The methods proposed in [2,6] have gained the plus point of previous knowledge of the position of the counter along with the background color which is green and decimal digits of red color.

2.3.1 Limitations

The use of these methods is limited because it will not work properly for every type of meter, and background color may also change when the reflection of light is going to change.

2.4 Connected Components Analysis (CCA)

Many approaches which are based on projection and color have been used for segmenting the digits[9]. With Connected Components Analysis (CCA), the usage of morphological operations for this purpose is considered [11].

2.4.1 Limitations

This method is limited because of its more dependency on results obtained from binarization and if the digits are broken and connected, it will not properly segment the digits.

2.5 Support Vector Machine (SVM)

A binary digit or non-digit method based on sliding window manner was proposed in [12] known as support vector machine. On the other hand, Gallo et al.[13] proposed Maximally Stable External Region (MSER).

2.5.1 Limitations

Because of the distortion and blurring in many images, the MSER algorithm cannot accurately and properly segments the digits.

2.6 Majchrak's Approach

There are several studies that deal with the automatic meter reading. For example, Majchrak et al.[14] propose a system using a camera, where the camera captures the image of the electric meter at uniform intervals. The system designed by Majchrak et al.[14] performs the pre-processing on the image by adjusting its brightness and contrast. After this adjustment, the image is finally cropped to acquire the numeric part. The algorithm used for the detection and segmentation of digits on pre-processed image of electrical meter is Support Vector Machine. Finally, the read digits of the output are transmitted to the server along with the details of customer, time and date etc.

2.6.1 Limitations

Although, the proposed framework by Majchrak et al.[14]is effective, but it increases the cost of the infrastructure as resources are required to automatically communicate the information.

2.7 Chandler's Approach

For digital instrumentation and reading the meter remotely, a method based on computer vision was proposed by Chandler et al.[16] in which a remote reading of the meter is done. The camera lens is placed near the energy meter panel, which captures the board image being downloaded to the monitor. For sharpening, detection of edges, segmenting the digits, and enhancing, the techniques which are based on image processing are applied to the recorded images. In this paper, the technique used to process the image are as follows: .

- Pre-processing of image, which includes processing of histogram, segmenting process, smoothing and calibration,
- The recognition of elements of meter image, which contains location of every digit and automatically recognition using algorithm of matching patterns for each digit's value.

2.7.1 Limitations

A fixed camera is required for each meter to capture the image. To implement this technique large amount of money is required. Further this scheme will only work in proposed scenario are prepossessing techniques are applied.

2.8 Binarization using Threshold

For AMR, Heydt et al.[17] described another prototype. According to this method, the picture captured through the camera was transmitted to the server PC by Zig-Bee where the picture goes through the process of segmenting the digits and identification by reading the numbers which were going to be used for the generation of bill. The device didn't use image processing for the execution of these steps. After the conversion of the picture to a grayscale picture, the

threshold was used for binarization, for the horizontally and vertically projection of numerical areas, identifying and segmenting the digits by using the algorithm of digit recognition.

2.8.1 Limitations

Because of the distortion in the picture, the digit loss increases. This distortion is caused by many factors like tilted angle, breaking of digit, reflection of lights on meter glass and dust on it. Therefore, this method of binarization is not accurate.

2.9 Vertical Edge Detection Algorithm (VEDA)

Another method is proposed by Barbose et al.[15] in which recognition of the digits of the electrical meter is done by the picture captured through a web camera by storing the outputs in the text file. For the conversion of the picture to binary picture, this method used the technique of adaptive thresholding extended by morphological operations. Before the images are segmented by using the algorithm of vertical edge detection, the resulted image is scanned. Then, the comparison between every segment of the picture and the stored template is done and the final output in the form of meter reading is saved in a text file.

2.9.1 Limitations

The proposed method by Barbose et al.[15] is not practical as the laptop's webcam and resources are used to process the image.

2.10 Long Short-term Memory (LSTM) network and Segmentation-free AMR

A bidirectional network for the recognition of the counter was designed by Geo et al.[1] named as Long Short-term Memory. According to this method, firstly, a sequence of the specific feature was generated by the network to combine the recurrent layers with convolutional layers. By following the representation of the specific feature, one digit is predicted by the attention decoder on each step. In the case of half digits, many errors were occurring but still, this method gave a

good accuracy. For the very first time, this was the single method presented by Gemoz et al.[18] which didn't use the technique of segmentation for automatic meter reading and without explicit detection of the counter, this method directly outputs the reading. In an end-to-end manner, a convolution neural network architecture was trained in which visual features from pictures are extracted by convolutional layers, and probabilities for every digit are extracted by fully connected layers.

2.10.1 Limitations

Although good accuracy was obtained by the network of LSTM, in the case of half digits, many errors occurred. By this algorithm of free segmentation, an impressive accuracy was obtained but the dataset used in this process was private and very large up to 180k pictures in the training phase. Moreover, these pictures were very fine and centered very well, and contains a clear portion of the image. Therefore, the images having a small size of meter with different capturing angles faced a lot of difficulties.

2.11 Conclusion

The approaches which have been developed for automatic meter reading are limited. Irrespective of the information of handcraft feature and private dataset, no author reported the computational time of their methods proposed because it is difficult for them to analyze their trade-off accuracy and speed of their application.

In contrast to the existing literature, the proposed project is developed for the scenario where resources are scarce. For example, it is not economical to replace all meters installed in Pakistan with AMR meters. As an alternate, the current project aims to explore smart ways of improving the effectiveness and efficiency of the current billing system by utilizing the existing infrastructure. To this, the proposed project will provide an image-based solution employing modern deep learning architectures which have been quite successful in similar computer vision applications including license plate recognition. The dataset examined in this project will be provided by our sectorial collaborator i.e., PITC. The successful delivery of this project will help the power sector of Pakistan to make the billing process more accurate and efficient with no additional cost. As the process will become more accurate without human errors, consumers will be able to pay the bills without delay and this will improve the efficiency of the currently deployed billing system.

Chapter 3

Methodology

An image-based automatic meter reading (AMR) approach incorporates two stages, to be specific as shown in Figure 3.1. .

- Counter detection
- Digit segmentation & recognition

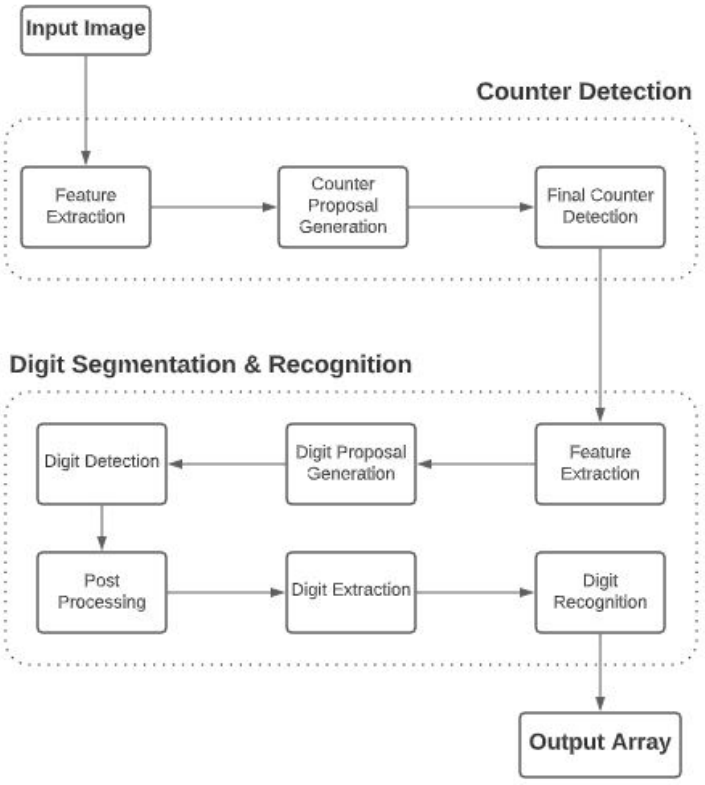

FIGURE 3.1: Work flow diagram of proposed technique for AMR system.

Counter detection is central to the entire AMR framework as it largely decides its accuracy and processing speed. Akin to other applications extracting textual information from images, e.g., license plate recognition, AMR is also considered an optical character recognition (OCR) problem. A plethora of studies has been performed to develop robust AMR frameworks employing conventional machine learning-based image processing tools. For example, for counter detection and digit segmentation, Shu et al.[21] employed the pixel-projection technique. Similarly, template matching techniques along with support vector machine classifiers have been employed for digit recognition [22]. Though impressive accuracy has been reported in various studies, still plenty has to be achieved before its real-time deployment. Recently, deep neural networks have gained much success across many computer vision applications. Despite the widespread success of these deep networks, the AMR domain has found only a few attempts.

Consistent with the literature, the proposed project will also employ modern deep learning schemes to develop a robust AMR pipeline. To start with, the initial step will be to collect raw images, of all types of meters installed in Pakistan, which will be used to make training, validation, and test datasets with a 90/10/10 cross-validation scheme using Python language and machine learning libraries. In the next stage, a multi-class classifier will be developed to determine the type of meter (out of 57 types of meters). Subsequently, a well-known You Only Look Once YOLO) based algorithm will be utilized as an object detector to detect the counter region of a meter. Next, a CNN-based counter recognition scheme will be designed to detect the meter reading. The entire pipeline will be trained, validated, and tested on a big dataset comprised of approx. 10k+ images. The trained model will be deployed in the circle-level central computer. A meter reader will take a snap of the meter from a mobile app, which will be automatically sent to the central computer (or server) for extraction of necessary information by simulating the trained model. The extracted information will be used for monthly billing. Figure 3.2 shows workflow of the proposed scheme.

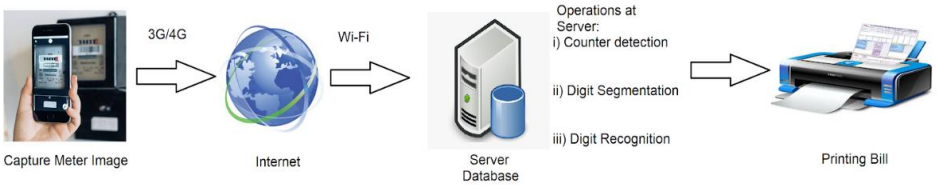

FIGURE 3.2: Demostration of the proposed framework for automatic meter reading.

For validation purposes, the entire framework will be evaluated in a sequential order. First, the manual annotation of images will be performed along with their quality check. Only those

images will be considered which will be readable by the human eye. The multi-class classifier will be tuned on the available data with the aim to achieve 100% accuracy. Once the multi-class classifier has been developed with acceptable accuracy, different variants of YOLO algorithm will be explored and optimized for counter detection. Being central to the whole framework, this stage will be ensured having maximum accuracy. Lastly, different digital recognition schemes will be evaluated to extract the relevant information e.g., meter ID and meter readings. The performance of all stages will be evaluated in quantitative terms e.g., accuracy(%).

3.1 Counter Detection

For the detection of the counter from the input image of the meter, firstly feature extraction will be done. The meter pictures have many textual blocks which can be confused with meter reading area block. The feature extraction is the process in which detection of counter is ensured on the basis of important features of region of interest (ROI) also known as counter. In this process, only one class is ensured to detect which is named as counter. The features for the extraction of the counter are the sharp four edges and curves in the form of rectangle around the strip of the meter reading which contains digits. After features extraction, the counter proposal generation would be done. The counter proposal generation includes the attributes of the counter position which ensures the detection of the counter using the values of these parameters. These parameters include center point (x, y), length, width and class number. By using these values, model detects the position of the counter by sliding a window on the pixel of the image.

3.2 Digit Segmentation & Recognition

After locating the counter in the input meter image, firstly digit detection will be done. In the detected counter of the image, each digit will be detected individually by using the process of features extraction. In this process, total eleven classes are ensured to detect. The features for the detection of each digits are the sharp edges around each digit within the region of detected counter. After feature extraction, digit proposal generation would be done. The digit proposal generation includes the parameters for each digit. Each digit will be detected firstly by making boundary boxes which are the features for each digit and depends on array of parameter defining the shape of the digit. This time parameter can be more than four because polygon shape will be drawn for each digit. After detecting the position of each digit, post processing will be done.

In this process each digit would be segmented by using the features of sharp edges of pattern of each digit. Now digit extraction will be done by using the values of the edge attributes of each digit pattern. By using these values for different digits in the counter area, digits will be extracted.

Chapter 4

Design and Implementation

Data set is the compulsory thing that is required when dealing with any machine learning technique, a significant portion of the overall accuracy of the system depends upon the selection of the data-set. The accurate and general the dataset the better result will occur.Second and most important thing is model. COMSATS Counter Detection (CCD) Model is designed and development for the detection of counter from the input meter image. How we created and designed this model is explained below along with its specifications and structure. For the second part of AMR system,to detect and recognize the image from the detect counter we designed and developed COMSATS Digit Recognition and Detection (CDRD) Model. Its specification, structure and features is also discussed in detail below.

4.1 Data-set

More than 2,000 images were used for the project which were taken from various consumer sites. As WAPDA serving more than 28 million people all over Pakistan, That is why project's dataset is comprised of different types of electric meters images taken under different scenarios (light, height, and angle). Figure 4.1 shows the variety of electric meters used for the project.

FIGURE 4.1: Diversity of meter types and conditions in WAPDA dataset.

One can see that .

- The counter part contains a small area of the electric meter which makes it difficult to detect,
- There are many written blocks on the meter with the serial numbers and specifications of the electric meter.

The dataset contains several images that are in poor condition. For example, some pictures are distorted, having broken glass, have low contrast due to sunlight, and some pictures having dirt on the screen. Moreover, the tilted angle of images due to meter reading creates problems in reading the digits from the meter.

Images were taken with the help of different mobile cameras such as Samsung Galaxy S21 Ultra, Vivo Y52 and Huawei P30 Lite. The format used to save the images was used JPG format and the resolution of these pictures was 3130 X 4140 pixels and 2350 X4150 pixels. The change in image resolution is due to the fact that different cameras were used to capture the images..

TABLE 4.1: Images captured with each camera.

Camera	Images
Galaxy S21 Ultra	950
Vivo Y52	580
Huawei P30 Lite	470
Total	2,000

Each counter of an electric meter in the dataset is comprised of different numbers ranging from 0-9. As it is known that a new meter always starts at zero which is why it takes a long time to convert its most significant digit, or it takes less time to convert its most significant digit. So, if we consider that each counter consists of eight digits, it means that 16,000 digits are manually annotated.

The dataset of 2000 images was divided into two parts. Out of 2000 images, 1600 images were used for training purposes, and 400 were used for testing purposes. For both training and testing, images are selected randomly. This technique is very useful for statistical analysis. This technique is not adopted by us but many other researchers also adopted the same division scheme to get more accurate and correct results.

4.2 COMSATS Counter Detection (CCD) Model

Recently in the field of object localization, the models based on YOLO(you only look once) [23] have demonstrated their worth, these models have achieved advanced and convincing results in the COCO and PASCAL VOC object detection competition [24]. Keeping in mind this we decided to make a COMSATS Counter Detection (CCD) model for the counter detection. In the proposed scheme, the main focus is on computational cost as there is only one class for detection, so a smart, advance, and smaller model is developed with fewer filters, less convolution, and pooling layers. So, we named our as COMSATS Counter Detection (CCD) and is shown in Figure 4.2.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

FIGURE 4.2: CCD network used to detect the counter region.

The CCD model contain fifty four convolution layers as shown in figure above. With in addition to convolution layer our model has Batch Normalization layer and LeakyRelu layer. All these layers are connected with each other like neurons are connected with each other in human body. The Figure 4.3 shows the physical representation of a single chunk of this model it is showing in which order these layer are connected. 24 chunk like this are connected to make CCD model. The Figure 4.4 demonstrate the summary of a small portion of the model. This summary is showing how many trainable and non trainable parameters are associated with each layer.

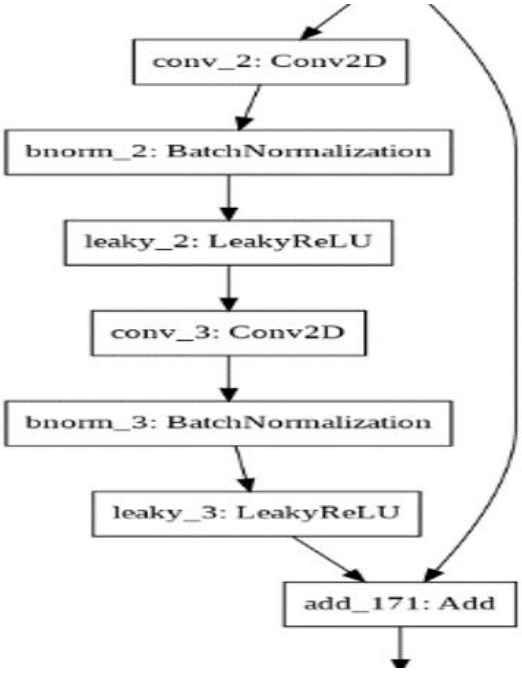

FIGURE 4.3: A chunk of CCD Model.

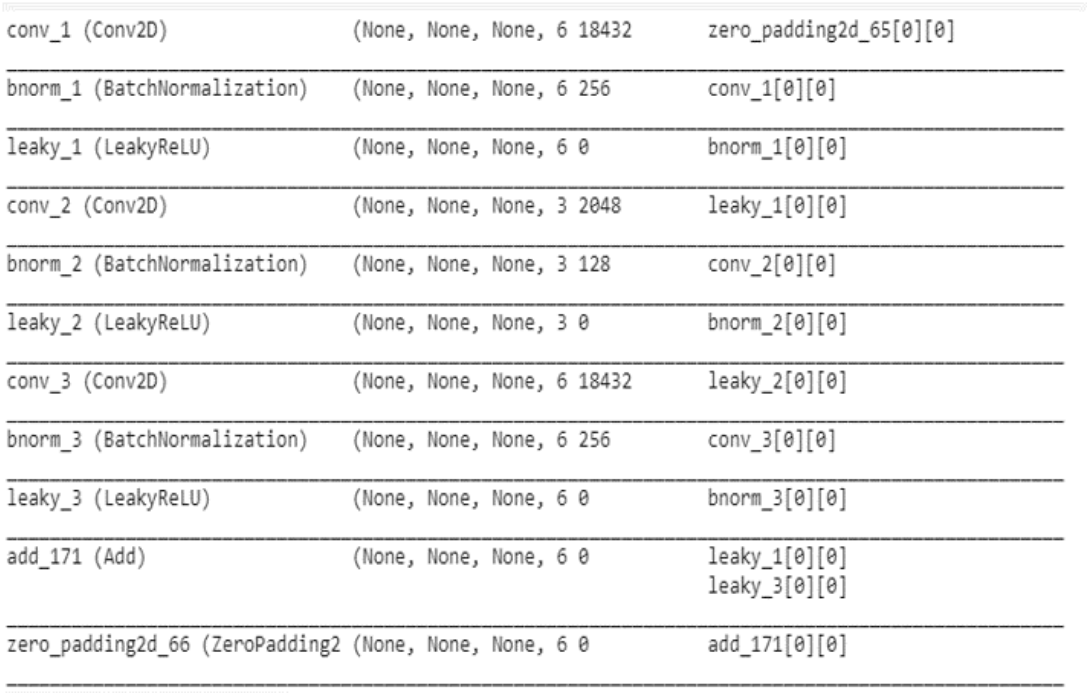

FIGURE 4.4: Summary of CCD Model.

For accessing the performance of counter detection in prediction we are using the intersection over union (IOU). The detection will be better if intersection over union will be close to one. There are two types of parameter which are .

- Trainable Parameter
- Non Trainable parameter

Parameters of neural network are typically the weights of the connections.when the input is transmitted between the neurons the weights are applied to the input along with bias. A model makes a function and then functions requires these parameter in order to make a prediction. So the model itself tunes these parameters during training.The trainable and non-trainable parameter associated with the CCD model are shown in Figure 7.4

FIGURE 4.5: Parameters of CCD Model.

4.3 COMSATS Digit Recognition and Detection (CDRD) Model

COMSATS Digit Recognition and Detection(CDRD) Model is more complex network than other models of counter detection. There are two outputs for digit recognition and detection model. Fist output is of class label while second output is of bounding-boxes, but in case of CDRD an additional branch is added that masks the output object, but it is apparent from previous outputs of CDRD. Thats why CDRD is totally a new idea. Furthermore, one of the key features of CDRD is pixel-to-pixel alignment but this feature is not present in the case of Faster R-CNN. Like Region Proposal Network (RPN), CDRD is also a two-stage procedure with the first stage identical while the second stage of both CDRD and Faster R-CNN is used to predict the bounding boxes and classes in parallel to detect the object. CDRD performs an additional function of binary masking for each RoI. This approach is much similar to the Fast R-CNN [11] in terms of their operations.

During training, multi-task loss function can be defined as:

$$L = L_{cls} + L_{box} + L_{mask} \tag{4.1}$$

where,

$$L_{cls} = \text{Classification loss}$$

$$L_{box} = \text{Boundary box loss}$$

$$L_{mask} = \text{Masking loss}$$

The classification and boundary box losses are the same as defined in [12] and masking loss is defined as "average binary cross-entropy loss" found by applying the technique of per-pixel sigmoid. This helps the model to create masks for every class without any distinction. Prediction of the class label is dependent on the classification branch which helps in the selection of output masks. Unlike Fully Convolutional Networks (FCNs) [25], it separates the prediction of mask and class which is a unique practice when applied to semantic segmentation. In the case of FCNs, there is a competition between masks and classes while in the case of CDRD there is no such competition between masks and classes. This technique is very useful for instance segmentation.

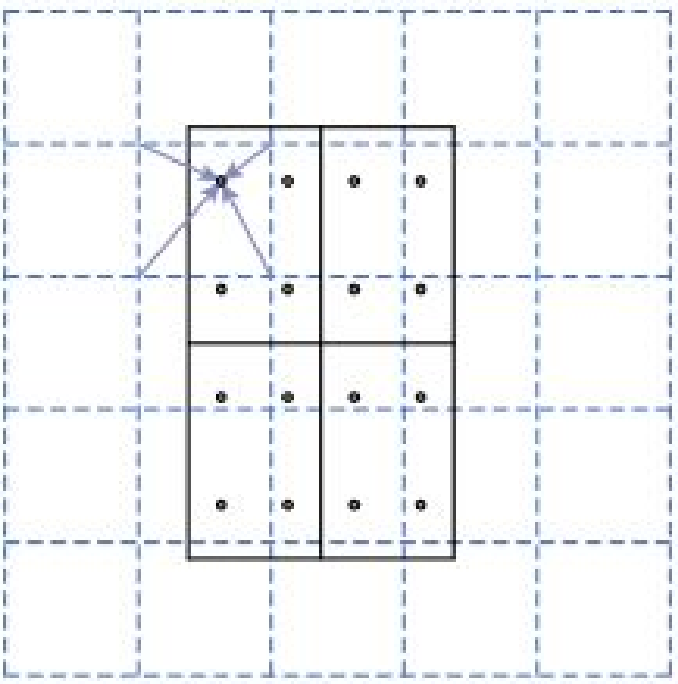

FIGURE 4.6: Characteristic map is depict by discontinuous grid. The solid line represent the Region of Intersection (ROI). Bilinear interpolation is used to calculate the value of sample point by ROI. Coordinates of sampling points and ROI are set free from quantization.

Instance segmentation is a complex technique because in this technique each instance has to be segmented precisely to make correct predictions of all the digits/objects present in an image. Therefore, this technique belongs to the classical computer vision techniques for the detection of objects where the task is to localize each object/digit using the bounding boxes. Figure 4.6 showing that good results can be achieved using the complex model of CDRD that is much faster, efficient, and accurate than other models of digit recognition and detection such for instance segmentation.

Instance segmentation is a complex technique because in this technique each instance must segmented precisely in order to make correct predictions of all the digits/objects present in an image. Therefore, this technique belongs to the classical computer vision techniques for the detection of objects where task is to localize each individual object/digit using the bounding boxes. Figure 5.6 showing that good results, can be achieved using complex model of CDRD that is much faster, efficient, and accurate than other models of digit recognition and detection such as instance segmentation.

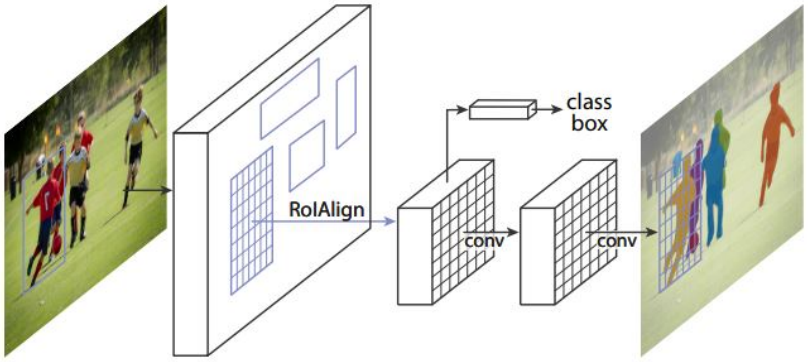

FIGURE 4.7: The CDRD framework for instance segmentation.

Here is the summary(of a small chunk) is showing how many trainable and non trainable parameters are associated with each layer, which layers are there, which layer is input and which one is output and and in which sequence they are connected as shown in Figure 4.8

Layer (type)	Output Shape	Param #	Connected to
input_image (InputLayer)	(None, None, None, 3	0	
zero_padding2d_1 (ZeroPadding2D	(None, None, None, 3	0	input_image[0][0]
conv1 (Conv2D)	(None, None, None, 6	9472	zero_padding2d_1[0][0]
bn_conv1 (BatchNorm)	(None, None, None, 6	256	conv1[0][0]
activation_1 (Activation)	(None, None, None, 6	0	bn_conv1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, None, None, 6	0	activation_1[0][0]
res2a_branch2a (Conv2D)	(None, None, None, 6	4160	max_pooling2d_1[0][0]
bn2a_branch2a (BatchNorm)	(None, None, None, 6	256	res2a_branch2a[0][0]
activation_2 (Activation)	(None, None, None, 6	0	bn2a_branch2a[0][0]
res2a_branch2b (Conv2D)	(None, None, None, 6	36928	activation_2[0][0]
bn2a_branch2b (BatchNorm)	(None, None, None, 6	256	res2a_branch2b[0][0]

FIGURE 4.8: The summary of CDRD Model (a chunk only) .

There are total 63,787,226 parameters associated with CDRD model which will using in making the function for the prediction. Out of these parameters there are 21,122,906 are trainable parameter which this model will learn using back propagation and the remaining are non-trainable as shown in Figure 4.9

```
Total params: 63,787,226
Trainable params: 21,122,906
Non-trainable params: 42,664,320
```

FIGURE 4.9: The parameters of CDRD Model.

Chapter 5

Counter Detection

Many alphabets and digits are written on different part of meter that can pose problems in meter reading by confusing. The second and most important thing is the counter position in the meter, in the whole meter image it constitutes a very small portion further the location of the counter changes as the meter type varies. In some meter it is located at the top, in some it is at the center whole in others it may be at any corner. There fore we are first going to locate and extract that counter from the meter image so that we can have maximum accuracy in the meter reading.

5.1 Working Principle

As the counter's reading is present on a small portion of the meter making it a little bit difficult to detect. So, it was decided to detect the counter first and then perform the segmentation of digits of the previously detected counter. To tackle both stages, Convolution Neural Networks (CNNs) are used. Remarkably, it is only the second work in which both stages are addressed using CNNs and the first one is being performed on Pakistan (WAPDA) datasets.

5.2 Counter Detection

Recently the models inspired by YOLO (you only look once) have proven their worth in the field of object detection because of their speed. They can execute 30-45 frames in just one minute [26]. That's why it is decided to make it fine-tune for counter detection purposes. With this in

mind, it was decided to develop a YOLO-like model for detecting the counter region. As the task at this point was to detect only one class and reduce the computational cost, so the decision was made to develop a smart and smaller model with few convolution layers and lesser filters. This model was named COMSATS Counter Detection (CCD). CCD yielded outstanding results through Intersection over Union (IoU) With an accuracy of 0.8. The formula for UoI is given below.

$$IOU = \left(\frac{area\left(\alpha_{pre} \cap \alpha_{actual}\right)}{area\left(\alpha_{pre} \cup \alpha_{actual}\right)} \right) \tag{5.1}$$

Where,

$$\alpha_{pre} = \text{Predicted bounding boxes}$$

$$\alpha_{actual} = \text{Ground truth bounding boxes}$$

As soon as the result of IoU approaches to 1, the efficiency of counter detection will be increased. Using these facts, it is believed that deep learning algorithms are not necessary if it is required to detect a single class of object.

First, anchor boxes are recalculated from the images of the meter. At this point, CCD starts to adjust the size of each nearest anchor to the size of the object in the input image. In the end, the number of filters is reduced in the last layer of CNN to generate a single output. The number of filters can be calculated using the following formula.

$$IOU = (C + 5) \times A \tag{5.2}$$

Where,

$$A = Number\ of\ anchor\ box$$

$$C = Number\ of\ classes$$

Next, multi-scale CCD training is employed. For training purposes, the labeled data is transferred to the model. CCD has comprised of a total of 106 layers out of which 3 layers are distinct. Firstly, the residual layer is used to forward activation to a deeper layer. In this layer, outputs of the first layer are added to the outputs of the second layer. The detection layer is the second layer which does detection at 3 different stages. The third layer is known as the up-sampling layer which is used to increase the spatial size of an image.

For output, the technique of IoU is applied over all bounding boxes and it then rejects the bounding boxes whose IoU value is greater than a threshold value. if two bounding boxes covering the same object then it will eliminate the bounding box having low probability. Once it is done then the algorithm starts to find the bounding box with the next highest class probabilities and performs the same process, the process repeats until covered with all bounding boxes.

5.3 Data Annotation

To train the model in deep learning both input and output are needed. The model will be based on the provided annotated dataset. In this case of AMR, the image of the meter is the input and the position of the counter is the output(counter contains meter reading). The position of the counter in the meter image is defined by the following four parameters. .

- Center x
- Center y
- Width
- Height

The required text files are created using a software routine named as “Labellmg”. Manually a rectangle is drawn against each picture in the dataset. Each image in the dataset has it own separate text file. The resulted text file contains above mentioned four parameters and the fifth parameter showing the class id which in our case always remain zero as we are dealing with only one single class (counter). The created text file is shown in Figure 5.1:

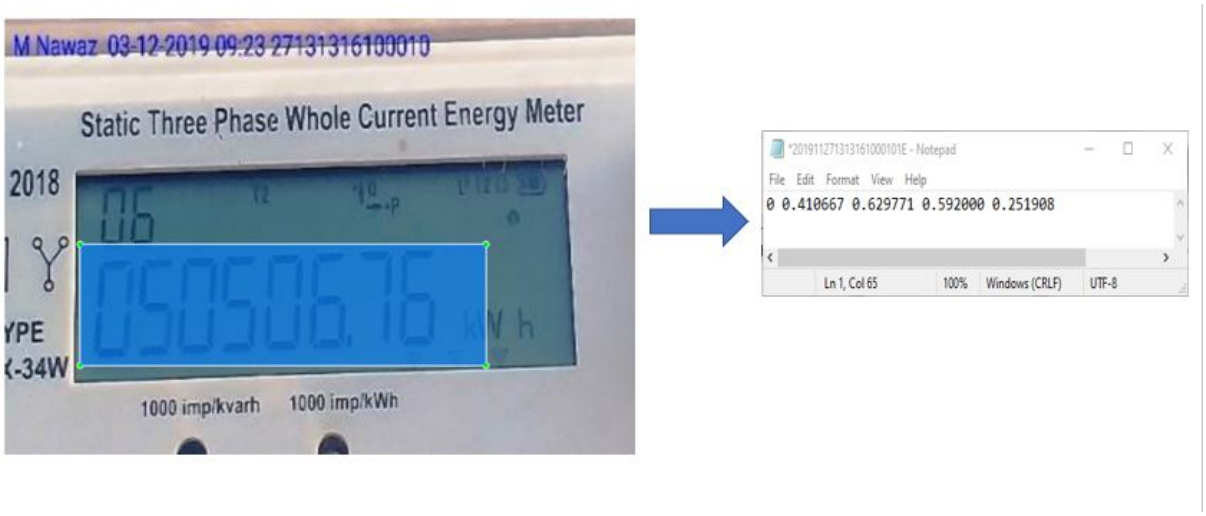

FIGURE 5.1: Meter image and its text file (annotation)

There are 2k pictures in the dataset (out of which 1600 are used for training and 400 are used for testing) and there are 2k text files one for each picture. Using these text files the CCD model is trained.

5.4 Results of CCD Model

The CCD model is then trained on the annotation on the Graphical Processing Unit (GPU) offered by the Google Colab, desktops cant be used for training because they don't have the required computational power they will go to sleep if were used for training. Model is trained on 2,000 iterations. After training the model on the GPU, the obtained weights were then downloaded for the purpose to run the algorithm on desktops. The results of the CCD Model is shown in Figure 5.2 and Figure 5.3:

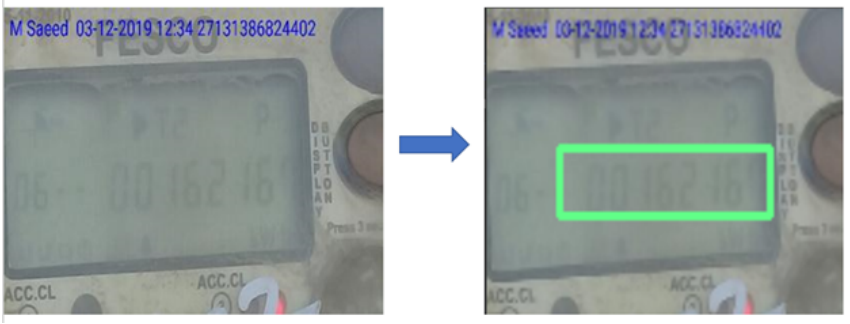

FIGURE 5.2: Counter detection on low resolution Picture..

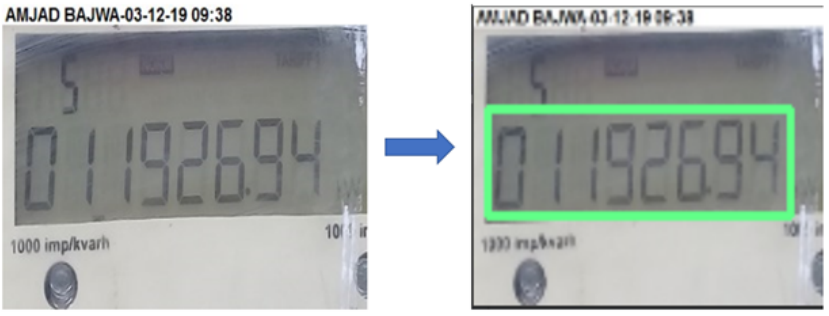

FIGURE 5.3: Counter detection on high resolution picture.

The CCD model is precisely detecting the counter in the meter image. Now the need is to extract the counter from the meter which is required to perform digit detection and recognition. The CCD model is now estimating the four coordinates which were given in annotated text file. So, the counterpart will be Extracted by using that four coordinates(center x, center y, width and height) as shown in figure 5.4.

FIGURE 5.4: Counter extraction by CCD model.

Chapter 6

Digit Segmentation & Recognition

Digit recognition involves three steps: .

- Digit detection,
- Digit segmentation,
- Digit recognition.

The first step is detecting the position of the digit in the image, the second stage involves segmenting (separating) each digit. The final and third step is about identifying the class of segmented digit whether it is 0,1,2,3...,9.

6.1 Working Principle

Counter was first detected to have better accuracy for digit detection and recognition because after locating the counter position in the meter image, the CDRD model will search for the digit in the specified area only. The output of CCD model is the input of the CDRD model. CDRD model will first extract the features of the digit by differentiate the digit and background. After detecting the features, the model will propose the area of digit then on the bases of that it will detect the digits. The detected digits are then preprocessed for better accuracy. In the second last step of the model's operation digit will be extracted and then these extracted digits will be recognized. The output of CDRD model will be an array containing the digits.

6.2 OCR Powered with Deep Learning

Optical Character Recognition (OCR) is a computer vision technique used to identify the different types of digits that are used in common mathematics. For different values of threshold, a string of digits was recognized by OCR from the meter image. First of all image processing is applied to the input meter image, so that OCR can work better as shown in figure 6.1:

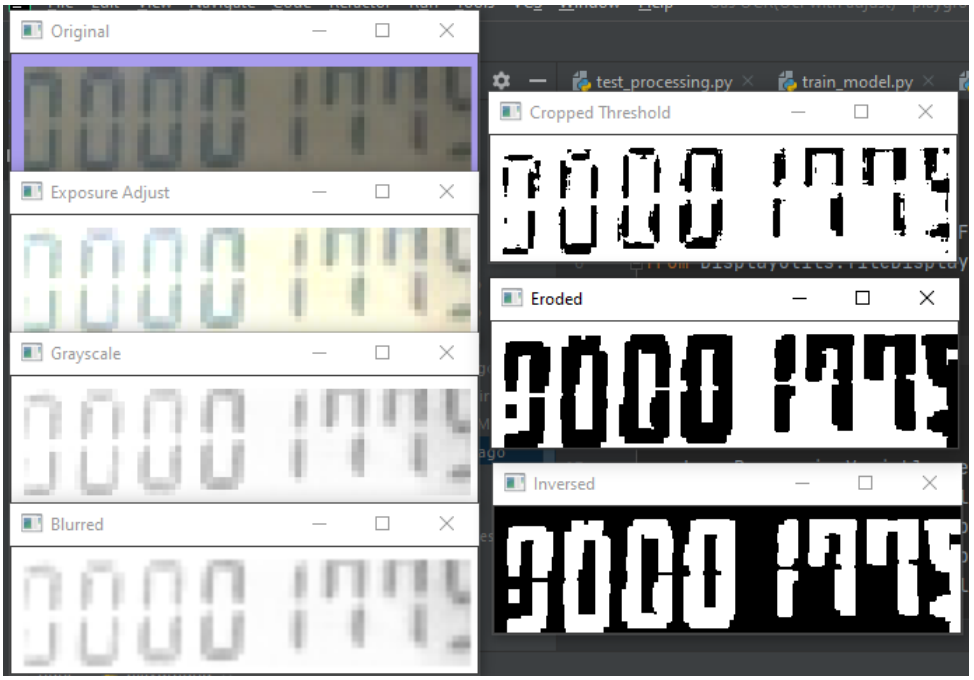

FIGURE 6.1: Enhancing the input image.

Then OCR Will detect the digits and draw a square box around each digit that it will detect. As after detecting the position of the digits in the picture with the help of OCR we will extract them. Each digit will be separated and then stored in a string as shown in the figure 6.2. After having the detected string of digit we will apply KNN (K Nearest Model) [28]. K in our case is 10 as we are going to have 10 objects from zero to nine. The string is passed to KNN which will recognize the digit whether it is 0,1,2,...,9. The results obtain from KNN is shown in Figure 6.3.

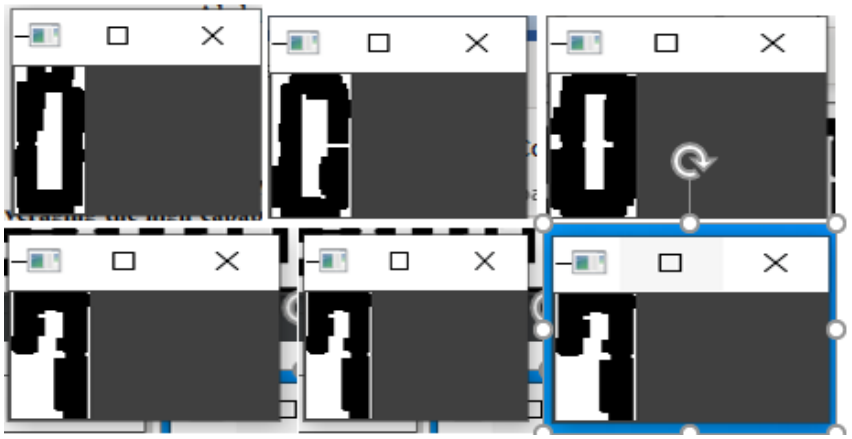

FIGURE 6.2: String of detected digits.

FIGURE 6.3: Recognized digits.

6.3 Tesseract

The text recognition based on open source is known as Tesseract [27]. In a large document, it is used for the recognition of the text from analysis of present layout. An image having single text line, it can be used with an external text detector. By calculating the width, height and center point for each digit, it firstly detects the digits by creating boundary boxes around each digit. Then, after detection of the digits, by using other functions it can read the digits as well. Here by using tesseract, we tried digit recognition in our case but didn't found much more accuracy. By using this technique, the results obtained are shown in Figure 6.4.

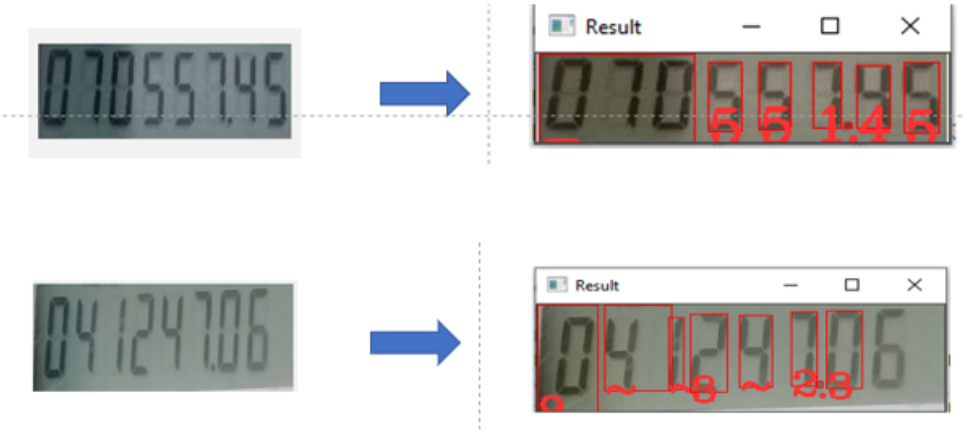

FIGURE 6.4: Digit detection and recognition by using Tesseract.

6.4 COMSATS Digit Recognition and Detection (CDRD) Model

It can be seen from our previous two techniques, which were unable to detect or recognize the digits from the detect counter due to some limitations: low picture quality, distorted image, half digits, light illumination factors, seven-segment display, and low contrast between the digits and background, and dust on the screen. Advances have been made in image processing and neural networks, but still, these techniques are prone to errors.

By taking into account the above-mentioned problems, the deep learning model is designed and developed named as COMSATS Digit Recognition and Detection (CDRD) Model. This model is based on Faster Recurrent Neural Network and Masking technology.

6.4.1 Digit Detection and Recognition

CDRD Model is employed in this technique to detect, recognize and mask the digit in the detected counter of meter image. After the detection of the counter, the next step is digit segmentation. For the digit recognition process, it is very important to segment the digits first which contains extracts of digits from the counter-image. Many factors such as dirt on meter screen, blurred image, rotation of digits, light reflection, broken glass, poor resolutions, numbering system, multi-colors, and noise increase the difficulty in the process of recognition of the digits. For the solution to these problems, an effective system is introduced which segments the digits. Before the recognition of a digit, each digit is firstly segmented from the counter and it is a very important part of digit recognition. The segmentation is divided into two parts. In the first part, because we have to detect each digit in the meter image, CDRD is used for the predictions based on sorting the digits and non-digits. In the second part, the digits which were very wide or too were flushed out. Depending upon the y-coordinates distance s, the extracted regions in the y-coordinate may vary with the same s amount. However, the following formula is representing the height of two consecutive digits:

$$S = |h_2 - h_1| \tag{6.1}$$

Many digits are needed to achieve good results. Therefore, we gather 11 classes, 10 objects (from digit 0–9), and one background. The reason to add background class is to differentiate between the digits and background effectively because we are working on seven segment display

where digit and background are oof similar color and with different light illumination(reflection) sometimes it becomes difficult to see by the naked eye.

6.4.2 Data Annotation for CDRD Model

There are different types of meters in the dataset, and each meter has several different digits. For example, there are nine digits in the one-meter type, 8 in the second, and 4 digits in the third-meter type. Most of the meter in the dataset has approximately 8 digits in the counterpart. So, approximately 16,000 digits were annotated for the training and test phase of the CDRD Model.

For the digit's annotations tool named VGG annotator [29] was used. A manual box was drawn against each digit in the meter. In the training of the CCD Model, each picture has its own separate file text file but this is not true in this annotation technique. In this part, there is two "Jason" files one of annotation of training dataset and the other file of a testing dataset.

One major difference between this annotation is that polygon shape boxes are drawn around each digit. The reason behind doing this is that, if we draw the square box across the digit then the box contains both digit and background and the model will not able to differentiate between the background and digit productively. So, to avoid this we draw the exact polygon shape of boxes around the digit. Polygon exactly encircles the digit only with precise shape. The annotation is shown in the Figure 6.5.

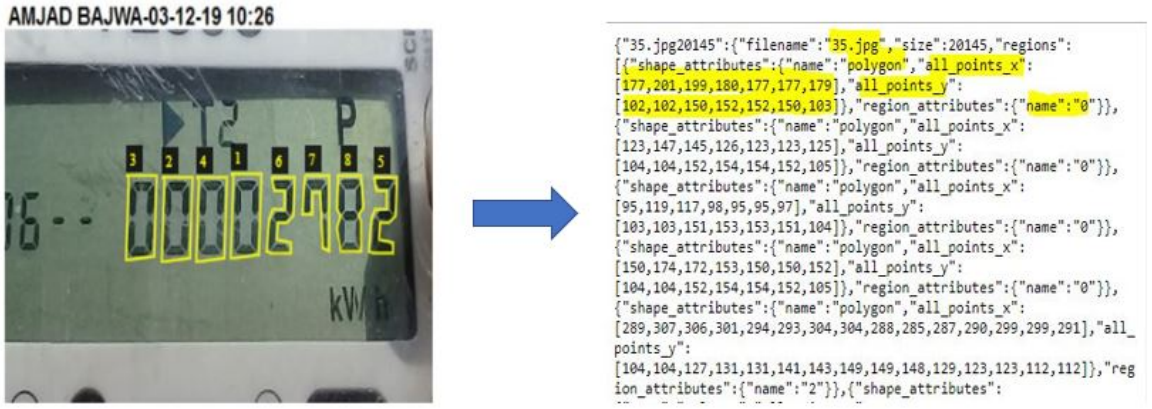

FIGURE 6.5: Meter image and its annotation.

As can be seen, exact boxes are drawn across each digit, the Jason file at the right-hand side shows the position of these digits in the image. First, it contains the picture name(35.jpg), then

the size of the image (20145), then the file show which shape is used(like polygon is used here), then the x and y coordinate, and at the end it shows to which digits (0) these coordinate belongs. That's how annotations are made for each picture in the data-set.

6.4.3 Results of CDRD Model

The CDRD model was trained on these annotations. The training is done on the Graphical processing unit offered by Google Colab. The model is trained on 100 epochs, with each epoch having 100 iterations. So, collectively model is trained on 10,000 iterations. The trained was then tested on the 400 pictures. The model is not only accurately predicting the digits in the meter, but it also returns that which pixels of the image belong to which digit. It masks each pixel of the single digit with a different color. The CDRD model is then tested to visualize the results. The pictures are passed to the CDRD model and the results obtained are shown in the Figure 6.6 and Figure 6.7.

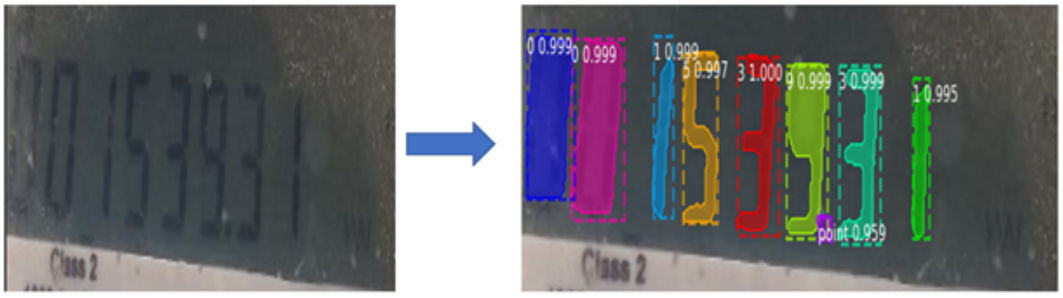

FIGURE 6.6: Digit detection and recognition (on blur image)

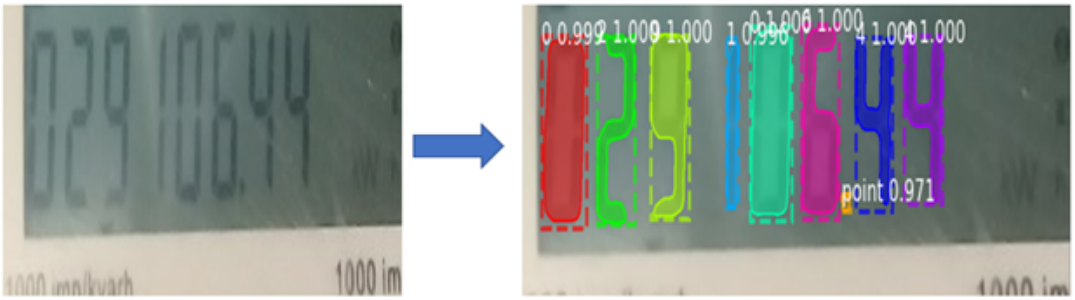

FIGURE 6.7: Digit detection and recognition (on clear image)

The detected and extracted counter is feed to the CDRD model to detect. Recognize and Mask the digits in the counterpart. As can be seen in the figure the model has accurately detected

the digits and mask them even though the pictures are not much clear and the resolution is not high. The breakthrough is in the detection of the point the developed model has accurately detect the small point which depicts how powerful the designed algorithm is. The model exactly detected the digit and print the detect class on the upper left side of each detected digit and on the upper right side of each digit the model is printing the precision with which it detected the digit. Like in figure (a) the first digit is zero so on the upper left side zero (0) is written while precision (0.999) is written on the upper right side (The model is estimating that it is about 99% sure that detected digit is zero).

Chapter 7

Results and Project Sustainability

First, in this chapter, the results of the proposed techniques will be discussed and then there is a brief description of the project sustainability by taking into account its environmental, social, and economical factors.

7.1 Results

Up to this point, the applied techniques are separately working well for counter and digits detection and recognition. The applied COMSATS Counter Detection (CCD) is efficiently detecting and extracting the counter from the meter image while on the other hand digits are detected, recognized, and masked by the COMSATS Digit Recognition and Detection (CDRD) Model. Figure 7.1 shows the result of CCD model while Figure 7.2 show the results of CDRD model.

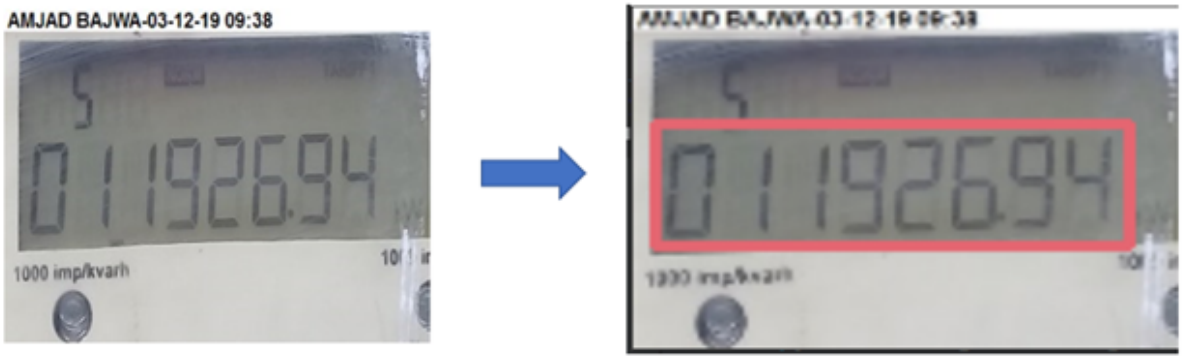

FIGURE 7.1: CCD model detecting the counter

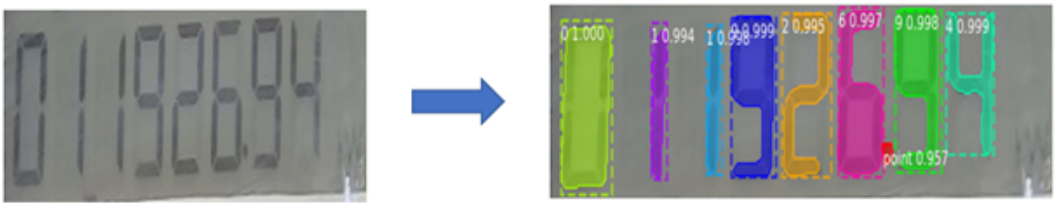

FIGURE 7.2: CDRD Model Results(Detecting, recognizing and masking the digits)

The CDRD model gives an array of the digits which are arranged on the base of its precision the digit with the highest precision will be on the first entries of an array while digits having the least precision will be on the last entries of an array. And there is a problem due to this, the whole sequence of the meter digits will be changed and the reading will be wrong can be seen in the Figure 7.3. A consumer might have to pay more than its consumption and there are equal chances that the user pays less than its usage. This problem is solved by sorting and making a dictionary. The model is then upgraded to return the position of the digit (x coordinates only) the digits will be arranged by using that x coordinates. The value of the x will be least for the first digit in the image while the value of the x will be high for the last digit in the image (as the value of the x increase if we move along the x-axis). So, a dictionary is made which saves the value of x and the corresponding digit. After this array is sort ascending order. When sorting is done pass that sorted array is then passed to the dictionary so that digit can be arranged in the sequence. This can be seen in the Figure 7.4:

FIGURE 7.3: Wrong sequence of digits

FIGURE 7.4: Correct sequence of digits

Now the need is to combine these two models to make a complete framework for meter detection and recognition. At first CCD model is employed which will detect the counter from the input meter image. Then CDRD model, the model will take CCD output as an input and perform its operation. As CDRD receives the counter position in the input so it will perform its operation on the area specified by the CCD and detect and recognize the digit and at last sorting arrays and dictionaries are placed to get the accurate arrangement of detected digits. The CCD and CDRD models are employed in successive order to make meter reading possible.

7.1.1 Accuracy Measures of CCD Model

The CCD model has performed well as it is very efficiently locating the counter in the image, further it outdoes the techniques applied in the literature, as shown in Table 7.1.

TABLE 7.1: Accuracy obtained by CCD and previous works found in the literature.

Approach	Training Accuracy	Testing Accuracy
Nodari et al. [19]	87.35%	70%
Gonc¸alves.[20]	90.2%	78.94%
Vanetti et al. [3]	92.5%	88.24%
CCD Model	**99.25%**	**98.07%**

The threshold value which is marked for intersection over union (IoU) is greater 0.7. The CCD done an outstanding job by yielding accuracy more than 0.9 for both data-sets which is greater than 0.7. Furthermore, it is noted that the detection with a lower IoU occurred mainly in cases where the meter/counter was inclined or tilted.

7.1.2 Loss Function of CCD Model

Loss Function is a method of evaluating how the well-designed algorithm is predicting. It can be fine-tuned to increase the accuracy of the designed model. If there is a big gap between the prediction and the actual results, then the loss function comes up with an error. To reduce the error, an optimization function is used whose function is to help the loss function in minimizing the prediction error. Figure 7.5 shows the loss function of the CCD model. It shows as the iteration level is increasing, the prediction error is going low. There will be high accuracy as the loss function approaches zero.

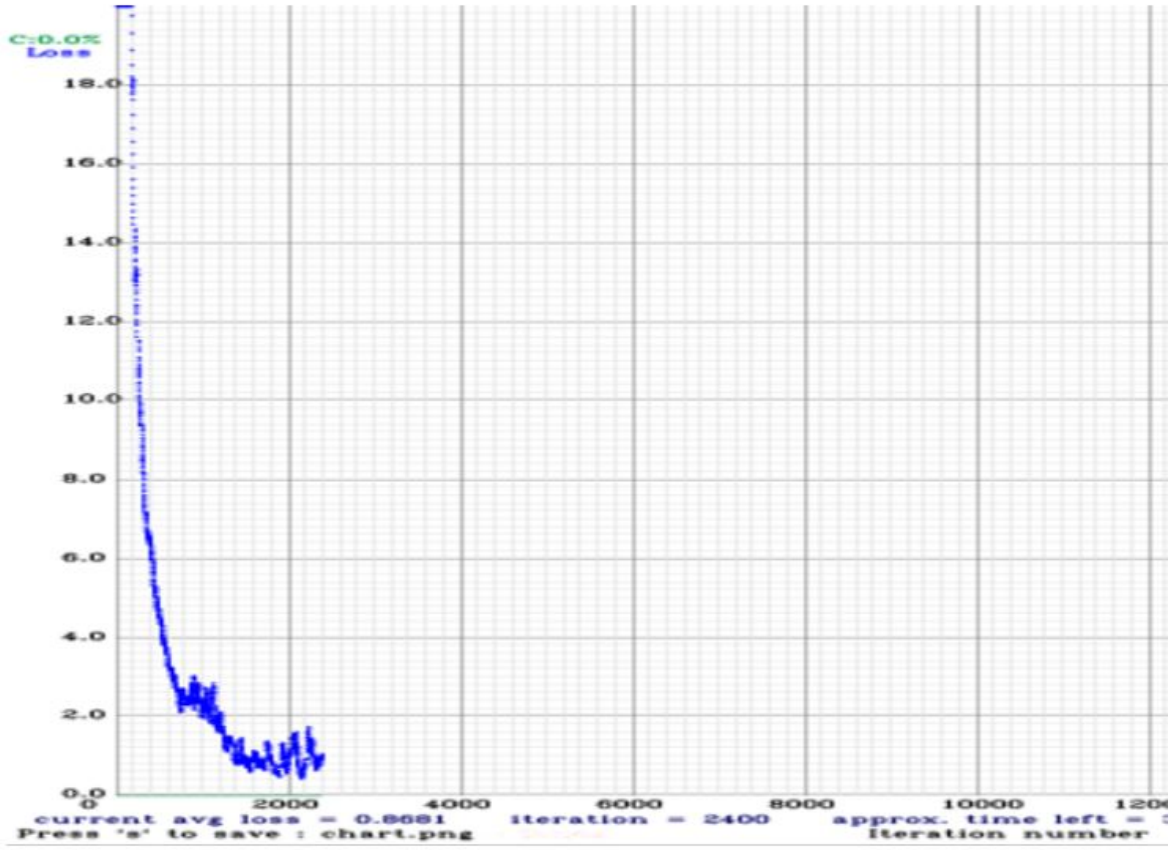

FIGURE 7.5: Loss Function of CCD Model.

7.1.3 Accuracy Measures of CDRD Model

The hyper-parameter that was initially set during training is no epoch, number of iteration, and learning rate. the model is trained on 100 epochs, 100 iterations per epoch (total 10,000 repetition), and a learning rate of 0.1,0.001 and 0.0001. The table 7.2 shows that our CDRD model has better accuracy among the three techniques discussed in the literature.

TABLE 7.2: Accuracy comparison of CDRD model and previous techniques

Approach	Training Accuracy	Testing Accuracy
Artificial intelligence Based OCR	58.1±1.69%	13.12±3.71%
Tesseract	42.5±2.54%	16.83±2.34%
Gallo et al.[19]	93.24%	80.13%
Fast Yolo.[20]	91.87%	78.33%
Vanetti et al. [13]	94.5%	81.45%
CDRD Model	**98.23±1.37%**	**96.8±1.23%**

It can be seen from the table 6.1 that our model has performed well among all the previous techniques. Gallo et al.[19], Vanetii et al.[13], and Fast Yolo.[20] the approach has far better accuracy than Artificial intelligence Based OCR and TESSART the only reason behind this is the difference in the data set. First, two techniques were applied to WAPDA dataset while the following three were applied to different datasets (The dataset has solid digits, and pictures were taken in a controlled environment).

7.1.4 Loss Function of CDRD Model

In the CDRD Model, there are four different kinds of losses: Region Proposal Network (RPN) (ground), RPN (forefront), CDRD (final output), and CDRD (box deterioration), which collectively give the loss as:

$$TotalLoss = (I_{cls} + I_{br})_{rpn} + (I_{cls} + I_{br})_{CDRD} \tag{7.1}$$

Where region suggestion loss for categorizing background and forefront is defined by, $(I_{cls} + I_{br})_{rpn}$ respectively. Classification between background of different classes is represented by, $\left(I_{cls(CDRD)}\right)$ where loss of refining bounding boxes is depict by $\left(I_{br(CDRD)}\right)$.

The graphical representation of the loss with each epoch is in the figure 7.6. The loss decay with each successive iteration.

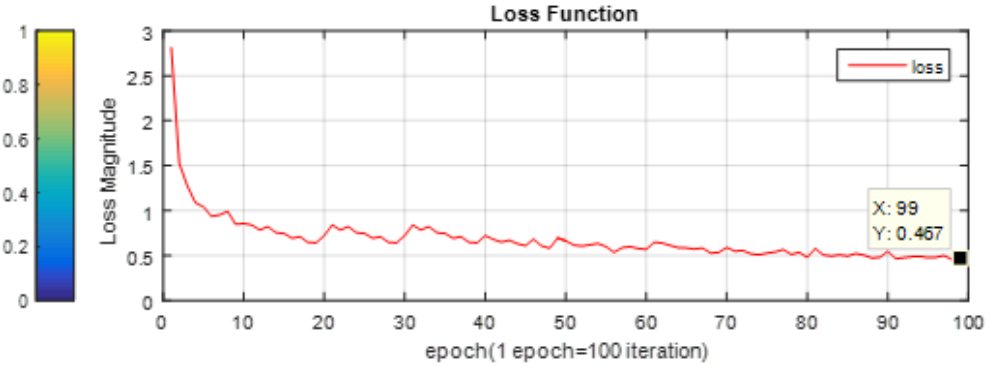

FIGURE 7.6: CDRD loss function

7.1.5 Web Application

A web application is developed as a user end product of the proposed scheme. The front end of the web app is designed in 'Android Studio' using HTML language and the back-end app is

developed in 'Pycharm' using python. A web page is developed in the 'Word Press'. This web application is linked with web page to make meter reading possible from anywhere. The meter reader will just open the link and choose their image from which he wants to read the meter reading. Both models are interfaced with web application. In the web application the user have to upload their meter image and the developed meter reading system will work to read that image. The web application is used in four steps. .

- Opening the Web Page
- Opening the website
- Choosing(uploading) the image, and
- Estimating (reading).

At first user have to open web page, which will lead the user to web application. Where the user will upload the meter image by clicking on 'choose'. After this 'Estimate!' button will pop up, by a click on that button the meter reading will be performed. The results are shown in the Figure 7.7-7.10 . This is how "Artificial Intelligence-based Automatic Meter Reading" will work.

FIGURE 7.7: Step 1:Lunching the web page

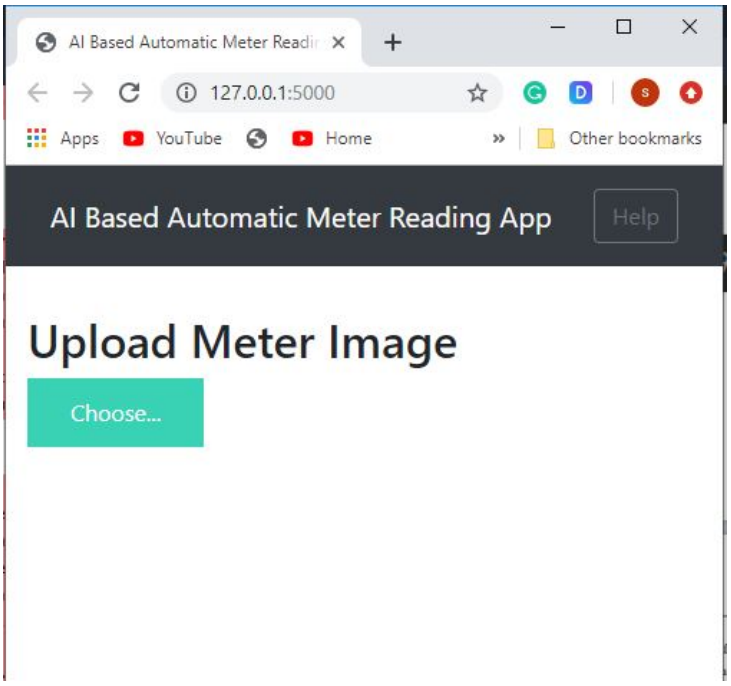

FIGURE 7.8: Step 1:Opening the Web App

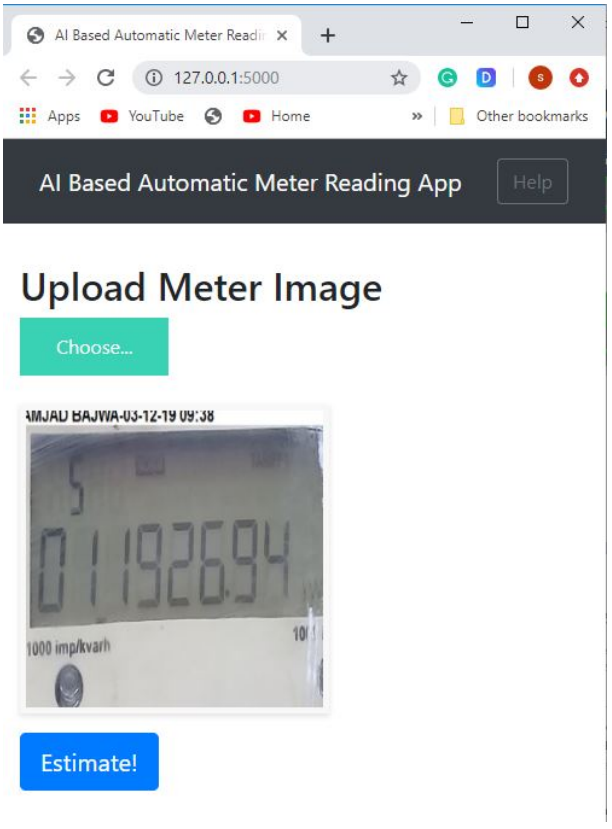

FIGURE 7.9: Step 2: Choosing an image

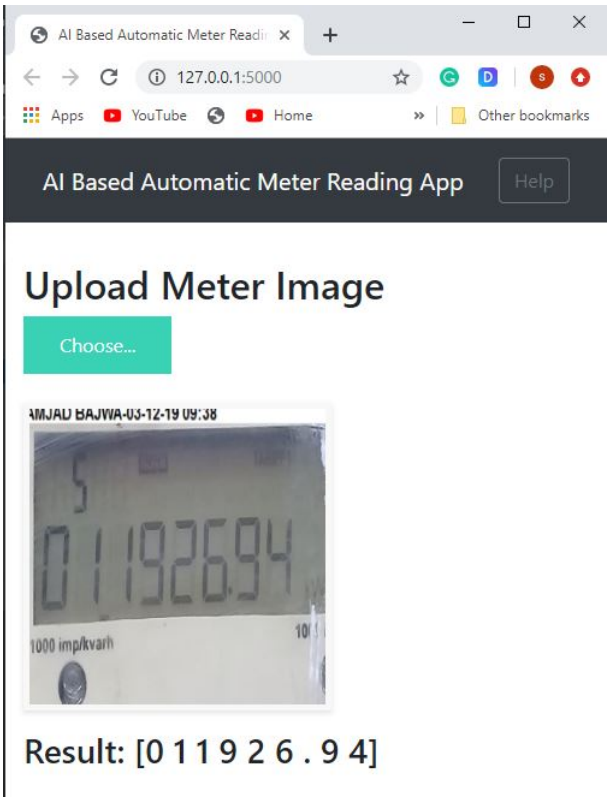

FIGURE 7.10: Step 3: Estimating.

7.1.6 Auto-mailing Electricity Bill

As soon as the web app will estimate the meter reading, an email will be sent to the consumer containing the bill in pdf format. A pdf page is designed through a python program which will take multiple things like consumer name, I'd, previous reading, and mailing address as inputs and make a pdf and save it in the current directory. The corresponding meter image is also placed in the bill as shown in Figure 7.11. When the bill will be generated, an automatic mailing system will be activated which will send that pdf containing the bill to the corresponding consumer as shown in Figure 7.12. So each time the meter reader will upload an image to a web app an automatic bill will be generated and sent to that particular consumer.

Dummy Electricity Bill

Consumer Name : Mazhar Farid

AMJAD BAJWA-03-12-19 09:38

01192694

1000 imp/kvarh

Figure 1:Meter Image

Meter id is : 74832

Previous Meter Reading is : 6019.5

Current Reading is : 11926.94

Unit Consumed : 5907.4400000000005

Price per unit : 24

Total bill is : 141778

Bill After Due(29June2021) 142078

Page 1/1

FIGURE 7.11: Electricity bill.

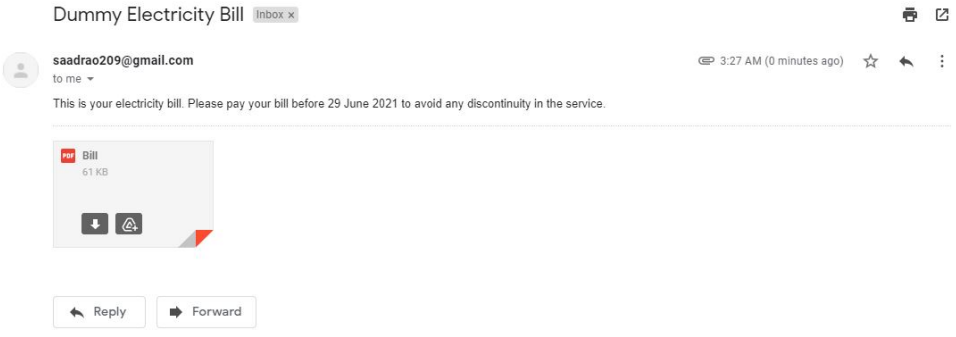

FIGURE 7.12: Received mail.

7.2 Security & Protection

The project is software-based. Therefore, there are no chances of physical hazards like electric shocks, etc. In the project, a web app is developed to read the electrical meter. Therefore, there are chances of hacking. The hacker can hack the web page and can manipulate the meter reading. Several steps regarding security are taken into account to avoid the above-mentioned situation.

7.2.1 Hosting

The first thing that matters towards website security is hosting. There are two types of hosting available one is shared and the other is dedicated. Dedicated have a specific IP address and not shared with anyone that's why it is more secure. It is not easy to attack that hosting.

7.2.2 Security Firewall

The second way to make the website more secure is by applying security firewall. WordPress has four to five built-in firewalls for the protection of the website. To prevent the manipulation of meter reading, a firewall would be used. Based on some security rules, a firewall is a device that provides network security by monitoring the incoming network traffic and outgoing traffic and then allows packets of data or block. The main purpose of a firewall is to make a hurdle between the incoming traffic which is coming from external sources like the internet and the internal network for blocking the malicious and harmful traffic like hackers and viruses. The security firewall which is applied for the protection in this website is the content delivery network (CDN). CDN generates a double-layer firewall to block the IP address of the hacker if he/she tries to hack admin access.

7.2.3 Optimized System

The system is properly optimized to avoid any back loops, holes, and bugs. Thirdly data is continuously being stored. If unfortunately, it hacks, at least data of the last 24 hours and later will be available.

7.3 Project Sustainability

Sustainability is one of the most important factors for any project. There is a high probability that the project will be failed If it is not sustainable. So, to make our project viable and sustainable we paid much attention to it. Best efforts are made to make sure that our project did not affect the environment in any way, and remain the cheapest solution to current problems. This proposed scheme not affecting the environment in any way but it is reducing human efforts to benefit humanity.

7.3.1 Environmental Sustainability

The first parameter to check the sustainability of the project is its impact on the environment. The end product of our project is a software routine (replacing the manual reading process with an automatic reading process) not a hardware structure, So there is no possibility that this project will release harmful gases into the air, deplete the ozone layer and cause global warming. This project has no harmful impacts on the wildlife and marine life but it is benefiting mankind in many ways as it is solving the problem of over-billing, reducing man efforts, and other problems associated with billing.

7.3.2 Economical Sustainability

The second parameter to check sustainability is to know either the project is economically Sustainable or not? The costs of resources or budget used for this project are quite affordable and reasonable to meet the desired specifications. As this project is present demand of WAPDA and is completed in a given time with a reasonable budget which makes the project economically sustainable or viable.

7.3.3 Social Sustainability

The third parameter is sustainability, which is about knowing either the project has positive or negative impacts on the people living within a particular society. This project presents the demand for WAPDA and has many positive impacts on society. Some of them are listed below: .

- The people's complaints of over-billing will be overcome,

- The utility companies will not face the problems of payment delays and additional processing, and
- It will increase the efficiency of bills and uplift the image of the power sector of Pakistan.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this paper, at both stages of counter detection and digits recognition a totally deep learning-based approaches are incorporated to have AMR system. WAPDA dataset is used in this scheme which consist of 2000 images of different meter types with picture taken in different scenarios (like light and height). The scholars have introduced several schemes to eliminate the problem associated with AMR. Still, all these techniques have their own limitations and disadvantages. All these problems are mitigated by incorporating highly effective and productive COMSATS Counter Detection (CCD) Model for counter detection and employed a deep learning model based on mask region CNN named as COMSATS Digit Recognition and Detection (CDRD) Model to curb the problem associated with detection, digit segmentation, and recognition. The incorporated CDRD model is able to effectively detect and segmentize the digits in all meter images and yield outclass recognition results for the given WAPDA dataset. The model not only recognize the digits but also mask each pixel associated with the individual digits with different color perfectly. Though there is a difference in the dataset, but accuracy obtained by this scheme is compared with the existing approaches. On the given dataset, results demonstrated that our employed scheme significantly beats state-of-the-art techniques in term of counter detection and digit recognition.

8.2 Future Work

The automatic electric meter reading project has bright future enhancement as well because of its trained deep learning algorithms of both detection and recognition process. The project is made for the WAPDA dataset to meet the demand of the WAPDA sector for automation in electricity reading which the meter has. However, the project can be extended on any dataset which includes the meter reading process by making efforts on training that dataset using the algorithms of the present project named COMSATS Counter Detection (CCD) and COMSATS Digit Recognition and Detection (CDRD). In addition to all these, the feedback from meter reading staff will also be considered for as a tool to further refine the proposed pipeline. In the future, the project can be extended in the following ways:

8.2.1 Automatic Gas Meter Reading

The project can be enhanced by making efforts for the automatic gas meter reading process. Firstly, the gas meter reading dataset will be collected from the gas distribution industry of Pakistan named as Oil and Gas Regulatory Authority (ORGD). This dataset will have the majority of the pictures of gas meters including different angles of clicked pictures and reflection of light on pictures. Then, the first step will be extracting the counter from the picture which includes meter reading by annotating the dataset for the training purpose using the CCD algorithm. After training, testing will be done and the counter of gas meter picture will be extracted. Now, from the extracted counters of the gas meter pictures, the digits of reading in the counter will be detected and recognized by using the CDRD algorithm. In this way, the project will be extended for the gas meter reading.

8.2.2 Automatic Water Meter Reading

Many digital water meters are used for the measurement of water intake from a few inches diameter of the pipe or water occupied by objects. In industries, the turbine water meters are the special ones used for the measurement of water intake quantity. So, this project can be extended by making efforts for the automatic water meter reading process. First of all, the water meter reading dataset will be collected from different industries which involve the use of water meters such as in boiler and sugar industries because they involve the use of water meters for the measurement of water quantity intake by the water pipes and in dams by the water turbines.

This dataset will have the majority of the pictures of water meters including different angles of clicked pictures and reflection of light on pictures. Then, the first step will be extracting the counter from the picture which includes meter reading by annotating the dataset for the training purpose using COMSATS Counter Detection (CCD) algorithm. After training, testing will be done and the counter of water meter picture will be extracted. Now, from the extracted counters of the water meter pictures, the digits of reading in the counter will be detected and recognized by using the COMSATS Digit Recognition and Detection (CDRD) algorithm. In this way, the project will be enhanced for the water meter reading. This is also an important extension of the automatic electric meter reading project.

Appendix A

CCD Model Code

```
def _conv_block(inp, convs, skip=True):
                x = inp
                count = 0
                for conv in convs:
                        if count == (len(convs) - 2) and skip:
                                skip_connection = x
                        count += 1
                        if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x)
                        # peculiar padding as darknet prefer left and top
                        x = Conv2D(conv['filter'],
                                           conv['kernel'],
                                           strides=conv['stride'],
                                           padding='valid' if conv['stride'] >
                                           1 else 'same', # peculiar padding
                                           as darknet prefer left and top
                                           name='conv_' + str(conv['layer_idx']),
                                           use_bias=False if conv['bnorm']
                                           else True)(x)
                        if conv['bnorm']: x =
                        BatchNormalization(epsilon=0.001, name='bnorm_' + str(conv['layer_idx']))(x)
                        if conv['leaky']: x =
                        LeakyReLU(alpha=0.1, name='leaky_' + str(conv['layer_idx']))(x)
                return add([skip_connection, x]) if skip else x


        def make_yolov3_model():
                input_image = Input(shape=(None, None, 3))
                # Layer  0 => 4
                x = _conv_block(input_image, [{'filter': 32, 'kernel': 3,

                'stride': 1, 'bnorm': True,
                'leaky': True,
```

```
'layer_idx': 0},
          {'filter': 64, 'kernel': 3,
          'stride': 2, 'bnorm': True,
          'leaky': True, 'layer_idx': 1},
           {'filter': 32, 'kernel': 1,
                  'stride': 1, 'bnorm': True,
                  'leaky': True, 'layer_idx': 2},
                          {'filter': 64, 'kernel': 3,
                          'stride': 1, 'bnorm': True,
                          'leaky': True, 'layer_idx': 3}])
# Layer  5 => 8
x = _conv_block(x, [{'filter': 128, 'kernel': 3, 'stride':
2, 'bnorm': True, 'leaky': True, 'layer_idx': 5},
        {'filter':  64, 'kernel': 1, 'stride':
 1, 'bnorm': True, 'leaky': True,
 'layer_idx': 6},
        {'filter': 128, 'kernel': 3, 'stride':
                1, 'bnorm': True, 'leaky': True,
         'layer_idx': 7}])
# Layer  9 => 11
x = _conv_block(x, [{'filter':  64, 'kernel': 1,
'stride': 1, 'bnorm': True, 'leaky': True,
'layer_idx': 9},
        {'filter': 128, 'kernel': 3, 'stride': 1,
        'bnorm': True, 'leaky': True,
        'layer_idx': 10}])
# Layer 12 => 15
x = _conv_block(x, [{'filter': 256, 'kernel': 3,
'stride': 2, 'bnorm': True, 'leaky': True,
'layer_idx': 12},
                {'filter': 128, 'kernel': 1,
                'stride': 1, 'bnorm': True, 'leaky': True, 'layer_idx': 13},
                {'filter': 256, 'kernel': 3, 'stride': 1,
                'bnorm': True, 'leaky': True, 'layer_idx': 14}])
# Layer 16 => 36
for i in range(7):
        x = _conv_block(x, [{'filter': 128, 'kernel': 1,
        'stride': 1, 'bnorm': True,
        'leaky': True,
        'layer_idx': 16+i*3},

        {'filter': 256, 'kernel': 3, 'stride': 1,
        'bnorm': True, 'leaky': True,
        'layer_idx': 17+i*3}])
skip_36 = x
# Layer 37 => 40
x = _conv_block(x, [{'filter': 512, 'kernel': 3,
'stride': 2, 'bnorm': True, 'leaky': True,
```

```
'layer_idx': 37},
                {'filter': 256, 'kernel': 1,
                'stride': 1, 'bnorm': True,
        'leaky': True, 'layer_idx': 38},
 {'filter': 512, 'kernel': 3, 'stride': 1,
 'bnorm': True, 'leaky': True,
 'layer_idx': 39}])
# Layer 41 => 61
for i in range(7):
        x = _conv_block(x, [{'filter': 256,
        'kernel': 1, 'stride': 1, 'bnorm': True,
        'leaky': True,
        'layer_idx': 41+i*3},
                {'filter': 512, 'kernel': 3, 'stride': 1,
                'bnorm': True, 'leaky': True,
                'layer_idx': 42+i*3}])
skip_61 = x
# Layer 62 => 65
x = _conv_block(x, [{'filter': 1024, 'kernel': 3,
'stride': 2, 'bnorm': True, 'leaky': True,
'layer_idx': 62},
 {'filter':  512, 'kernel': 1, 'stride': 1,
 'bnorm': True, 'leaky': True, 'layer_idx': 63},
        {'filter': 1024, 'kernel': 3,
        'stride': 1, 'bnorm': True, 'leaky': True,
        'layer_idx': 64}])
# Layer 66 => 74
for i in range(3):
        x = _conv_block(x, [{'filter':  512,
        'kernel': 1, 'stride': 1, 'bnorm': True,
        'leaky': True,
        'layer_idx': 66+i*3},
                {'filter': 1024, 'kernel': 3, 'stride': 1,
                'bnorm': True, 'leaky': True,
                'layer_idx': 67+i*3}])
# Layer 75 => 79
x = _conv_block(x, [{'filter':  512,
'kernel': 1,
'stride': 1, 'bnorm': True, 'leaky': True,
'layer_idx': 75},
        {'filter': 1024, 'kernel': 3,
                'stride': 1, 'bnorm': True,
                'leaky': True, 'layer_idx': 76},
        {'filter':  512, 'kernel': 1, 'stride': 1,
        'bnorm':        {'filter': 1024, 'kernel': 3,
        'stride': 1, 'bnorm': True, 'leaky': True,
        'layer_idx': 78},
                {'filter':  512, 'kernel': 1, 'stride': 1,
```

```
                'bnorm': True, 'leaky': True,
                'layer_idx': 79}], skip=False)
# Layer 80 => 82
yolo_82 = _conv_block(x, [{'filter': 1024,
'kernel': 3, 'stride': 1, 'bnorm': True,
'leaky': True,  'layer_idx': 80},
          {'filter':  255, 'kernel': 1, 'stride': 1,
          'bnorm': False, 'leaky': False,
          'layer_idx': 81}], skip=False)
# Layer 83 => 86
x = _conv_block(x, [{'filter': 256,
'kernel': 1, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 84}], skip=False)
x = UpSampling2D(2)(x)
x = concatenate([x, skip_61])
# Layer 87 => 91
x = _conv_block(x, [{'filter': 256,
'kernel': 1,
'stride': 1, 'bnorm': True, 'leaky': True,
'layer_idx': 87},
 {'filter': 512, 'kernel': 3, 'stride': 1,
 'bnorm': True, 'leaky': True,
 'layer_idx': 88},
        {'filter': 256, 'kernel': 1,
        'stride': 1, 'bnorm': True, 'leaky': True,
        'layer_idx': 89},
 {'filter': 512, 'kernel': 3, 'stride': 1,
 'bnorm': True, 'leaky': True,
 'layer_idx': 90},
                                        {'filter': 256,
                                        'kernel': 1, 'stride': 1,
                'bnorm': True, 'leaky': True,
                'layer_idx': 91}], skip=False)
# Layer 92 => 94
yolo_94 = _conv_block(x, [{'filter': 512,
'kernel': 3, 'stride': 1, 'bnorm': True,
'leaky': True,  'layer_idx': 92},
  {'filter': 255, 'kernel': 1, 'stride': 1,
  'bnorm': False, 'leaky': False,
  'layer_idx': 93}], skip=False)
# Layer 95 => 98
x = _conv_block(x, [{'filter': 128, 'kernel': 1,
'stride': 1, 'bnorm': True, 'leaky': True,
'layer_idx': 96}], skip=False)
x = UpSampling2D(2)(x)
x = concatenate([x, skip_36])
# Layer 99 => 106
yolo_106 = _conv_block(x, [{'filter': 128,
```

```
        'kernel': 1, 'stride': 1, 'bnorm': True,
        'leaky': True,  'layer_idx': 99},
           {'filter': 256, 'kernel': 3, 'stride': 1,
           'bnorm': True,  'leaky': True,
           'layer_idx': 100},
    {'filter': 128, 'kernel': 1, 'stride': 1,
    'bnorm': True,  'leaky': True,
    'layer_idx': 101},
         {'filter': 256, 'kernel': 3, 'stride': 1,
         'bnorm': True,  'leaky': True,  'layer_idx': 102},
                   {'filter': 128, 'kernel': 1,
                   'stride': 1, 'bnorm': True,
                   'leaky': True,  'layer_idx': 103},
              {'filter': 256, 'kernel': 3,
              'stride': 1, 'bnorm': True,  'leaky': True,  'layer_idx': 104},
          {'filter': 255, 'kernel': 1,
          'stride': 1, 'bnorm': False,
          'leaky': False, 'layer_idx': 105}], skip=False)
        model = Model(input_image, [yolo_82, yolo_94, yolo_106])
        return model
```

Appendix B

CCD Training Code

```
# -*- coding: utf-8 -*-
"""CCD_.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1poBMScB7Gbfu1A_5uee8MhRzKXjpzWzH

**Connect google drive**
"""

# Check if NVIDIA GPU is enabled
!nvidia-smi

from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive

"""**1) Clone the Darknet**


"""

!git clone https://github.com/AlexeyAB/darknet

"""**2) Compile Darknet using Nvidia GPU**

"""

# Commented out IPython magic to ensure Python compatibility.
# change makefile to have GPU and OPENCV enabled
```

```
# %cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!make

"""**3) Configure Darknet network for training YOLO V3**"""

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

!sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
!sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 4000/' cfg/yolov3_training.cfg
!sed -i '610 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=1@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=18@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=18@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=18@' cfg/yolov3_training.cfg

# Create folder on google drive so that we can save there the weights
!mkdir "/mydrive/yolov3"

!echo "Koala" > data/obj.names
!echo -e 'classes= 1\ntrain  = data/train.txt\nvalid  = data/test.txt\nnames = data/obj.names
\nbackup = /mydrive/yolov3' > data/obj.data
!mkdir data/obj

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

"""**4) Extract Images**

The images need to be inside a zip
archive called "images.zip" and they need to
be inside the folder "yolov3" on Google Drive
"""

!unzip /mydrive/yolov3/images.zip -d data/obj

# We're going to convert the class
index on the .txt files. As we're working with
only one class, it's supposed to be class 0.
# If the index is different from 0 then we're
going to change it.
import glob
import os
import re
```

```
txt_file_paths = glob.glob(r"data/obj/*.txt")
for i, file_path in enumerate(txt_file_paths):
    # get image size
    with open(file_path, "r") as f_o:
        lines = f_o.readlines()

        text_converted = []
        for line in lines:
            print(line)
            numbers = re.findall("[0-9.]+", line)
            print(numbers)
            if numbers:

                # Define coordinates
                text = "{} {} {} {} {}".format(0, numbers[1], numbers[2], numbers[3], numbers[4])
                text_converted.append(text)
                print(i, file_path)
                print(text)
        # Write file
        with open(file_path, 'w') as fp:
            for item in text_converted:
                fp.writelines("%s\n" % item)

import glob
images_list = glob.glob("data/obj/*.jpg")
print(images_list)

#Create training.txt file
file = open("data/train.txt", "w")
file.write("\n".join(images_list))
file.close()

"""**6) Start the training**"""

# Start the training
!./darknet detector train data/obj.data cfg/yolov3_training.cfg darknet53.conv.74 -dont_show
```

Appendix C

CCD Testing Code

```
import cv2
import numpy as np
import glob
import random
import os

# Load Yolo
net = cv2.dnn.readNet("yolov3_training_2000.weights", "yolov3_testing.cfg")
path=r'C:\Users\Saad Masrur Rao\Desktop\jpg'
# Name custom object
classes = [""]

# Images path
images_path = sorted(glob.glob(r"C:\Users\Saad Masrur Rao\Desktop\Sirdatatesting\\*.jpg" ))




layer_names = net.getLayerNames()
output_layers = [layer_names[i[0] - 1] for i in net.getUnconnectedOutLayers()]
colors = np.random.uniform(0, 255, size=(len(classes), 3))

# Insert here the path of your images
random.shuffle(images_path)
# loop through all the images
d=83;
for img_path in images_path:
    # Loading image

    d=d+1;
    img = cv2.imread(img_path)
```

```
cv2.imshow('Image', img)
cv2.waitKey(0)
print(img.shape)
print(d)



img = cv2.resize(img, None, fx=0.4, fy=0.4)
height, width, channels = img.shape

# Detecting objects
blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0), True, crop=False)

net.setInput(blob)
outs = net.forward(output_layers)

# Showing informations on the screen
class_ids = []
confidences = []
boxes = []

for out in outs:
    for detection in out:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.3:
            # Object detected
            print(class_id)
            center_x = int(detection[0] * width)
            center_y = int(detection[1] * height)
            w = int(detection[2] * width)
            h = int(detection[3] * height)

            # Rectangle coordinates
            x = int(center_x - w / 2)
            y = int(center_y - h / 2)

            boxes.append([x, y, w, h])
            confidences.append(float(confidence))
            class_ids.append(class_id)

indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
print(indexes)
font = cv2.FONT_HERSHEY_PLAIN
for i in range(len(boxes)):
    if i in indexes:
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
```

```
            color = colors[class_ids[i]]
            cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
            cv2.putText(img, label, (x, y + 30), font, 3, color, 2)

    crop = img[y:y + h, x:x + w]
    cv2.imwrite(os.path.join(path, 'meter_'+str(d)+'.jpg'),crop )
    img = cv2.resize(crop, (100 , 100))
    cv2.imshow('Image', img)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

Appendix D

CDRD Model Code

```
import os
import random
import datetime
import re
import math
import logging
from collections import OrderedDict
import multiprocessing
import numpy as np
import tensorflow as tf
import keras
import keras.backend as K
import keras.layers as KL
import keras.engine as KE
import keras.models as KM

from mrcnn import utils

# Requires TensorFlow 1.3+ and Keras 2.0.8+.
from distutils.version import LooseVersion
assert LooseVersion(tf.__version__) >= LooseVersion("1.3")
assert LooseVersion(keras.__version__) >= LooseVersion('2.0.8')


############################################################
#  Utility Functions
############################################################

def log(text, array=None):
    """Prints a text message. And, optionally, if a Numpy array is provided it
```

```
    prints it's shape, min, and max values.
    """
    if array is not None:
        text = text.ljust(25)
        text += ("shape: {:20}  ".format(str(array.shape)))
        if array.size:
            text += ("min: {:10.5f}  max: {:10.5f}".format(array.min(),array.max()))
        else:
            text += ("min: {:10}  max: {:10}".format("","))
        text += "  {}".format(array.dtype)
    print(text)


class BatchNorm(KL.BatchNormalization):
    """Extends the Keras BatchNormalization class to allow a central place
    to make changes if needed.

    Batch normalization has a negative effect on training if batches are small
    so this layer is often frozen (via setting in Config class) and functions
    as linear layer.
    """
    def call(self, inputs, training=None):
        """
        Note about training values:
            None: Train BN layers. This is the normal mode
            False: Freeze BN layers. Good when batch size is small
            True: (don't use). Set layer in training mode even when making inferences
        """
        return super(self.__class__, self).call(inputs, training=training)


def compute_backbone_shapes(config, image_shape):
    """Computes the width and height of each stage of the backbone network.

    Returns:
        [N, (height, width)]. Where N is the number of stages
    """
    if callable(config.BACKBONE):
        return config.COMPUTE_BACKBONE_SHAPE(image_shape)

    # Currently supports ResNet only
    assert config.BACKBONE in ["resnet50", "resnet101"]
    return np.array(
        [[int(math.ceil(image_shape[0] / stride)),
            int(math.ceil(image_shape[1] / stride))]
            for stride in config.BACKBONE_STRIDES])
```

```
############################################################
#  Resnet Graph
############################################################

# Code adopted from:
# https://github.com/fchollet/deep-learning-models/blob/master/resnet50.py

def identity_block(input_tensor, kernel_size, filters, stage, block,
                   use_bias=True, train_bn=True):
    """The identity_block is the block that has no conv layer at shortcut
    # Arguments
        input_tensor: input tensor
        kernel_size: default 3, the kernel size of middle conv layer at main path
        filters: list of integers, the nb_filters of 3 conv layer at main path
        stage: integer, current stage label, used for generating layer names
        block: 'a','b'..., current block label, used for generating layer names
        use_bias: Boolean. To use or not use a bias in conv layers.
        train_bn: Boolean. Train or freeze Batch Norm layers
    """
    nb_filter1, nb_filter2, nb_filter3 = filters
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = KL.Conv2D(nb_filter1, (1, 1), name=conv_name_base + '2a',
                  use_bias=use_bias)(input_tensor)
    x = BatchNorm(name=bn_name_base + '2a')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
                  name=conv_name_base + '2b', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2b')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c',
                  use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2c')(x, training=train_bn)

    x = KL.Add()([x, input_tensor])
    x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
    return x


def conv_block(input_tensor, kernel_size, filters, stage, block,
               strides=(2, 2), use_bias=True, train_bn=True):
    """conv_block is the block that has a conv layer at shortcut
    # Arguments
        input_tensor: input tensor
        kernel_size: default 3, the kernel size of middle conv layer at main path
```

```
        filters: list of integers, the nb_filters of 3 conv layer at main path
        stage: integer, current stage label, used for generating layer names
        block: 'a','b'..., current block label, used for generating layer names
        use_bias: Boolean. To use or not use a bias in conv layers.
        train_bn: Boolean. Train or freeze Batch Norm layers
    Note that from stage 3, the first conv layer at main path is with subsample=(2,2)
    And the shortcut should have subsample=(2,2) as well
    """
    nb_filter1, nb_filter2, nb_filter3 = filters
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = KL.Conv2D(nb_filter1, (1, 1), strides=strides,
                  name=conv_name_base + '2a', use_bias=use_bias)(input_tensor)
    x = BatchNorm(name=bn_name_base + '2a')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
                  name=conv_name_base + '2b', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2b')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.Conv2D(nb_filter3, (1, 1), name=conv_name_base +
                  '2c', use_bias=use_bias)(x)
    x = BatchNorm(name=bn_name_base + '2c')(x, training=train_bn)

    shortcut = KL.Conv2D(nb_filter3, (1, 1), strides=strides,
                         name=conv_name_base + '1', use_bias=use_bias)(input_tensor)
    shortcut = BatchNorm(name=bn_name_base + '1')(shortcut, training=train_bn)

    x = KL.Add()([x, shortcut])
    x = KL.Activation('relu', name='res' + str(stage) + block + '_out')(x)
    return x


def resnet_graph(input_image, architecture, stage5=False, train_bn=True):
    """Build a ResNet graph.
        architecture: Can be resnet50 or resnet101
        stage5: Boolean. If False, stage5 of the network is not created
        train_bn: Boolean. Train or freeze Batch Norm layers
    """
    assert architecture in ["resnet50", "resnet101"]
    # Stage 1
    x = KL.ZeroPadding2D((3, 3))(input_image)
    x = KL.Conv2D(64, (7, 7), strides=(2, 2), name='conv1', use_bias=True)(x)
    x = BatchNorm(name='bn_conv1')(x, training=train_bn)
    x = KL.Activation('relu')(x)
    C1 = x = KL.MaxPooling2D((3, 3), strides=(2, 2), padding="same")(x)
```

```
    # Stage 2
    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1), train_bn=train_bn)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='b', train_bn=train_bn)
    C2 = x = identity_block(x, 3, [64, 64, 256], stage=2, block='c', train_bn=train_bn)
    # Stage 3
    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', train_bn=train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='b', train_bn=train_bn)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='c', train_bn=train_bn)
    C3 = x = identity_block(x, 3, [128, 128, 512], stage=3, block='d', train_bn=train_bn)
    # Stage 4
    x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a', train_bn=train_bn)
    block_count = {"resnet50": 5, "resnet101": 22}[architecture]
    for i in range(block_count):
        x = identity_block(x, 3, [256, 256, 1024], stage=4, block=chr(98 + i), train_bn=train_bn)
    C4 = x
    # Stage 5
    if stage5:
        x = conv_block(x, 3, [512, 512, 2048], stage=5, block='a', train_bn=train_bn)
        x = identity_block(x, 3, [512, 512, 2048], stage=5, block='b', train_bn=train_bn)
        C5 = x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c', train_bn=train_bn)
    else:
        C5 = None
    return [C1, C2, C3, C4, C5]


############################################################
#  Proposal Layer
############################################################

def apply_box_deltas_graph(boxes, deltas):
    """Applies the given deltas to the given boxes.
    boxes: [N, (y1, x1, y2, x2)] boxes to update
    deltas: [N, (dy, dx, log(dh), log(dw))] refinements to apply
    """
    # Convert to y, x, h, w
    height = boxes[:, 2] - boxes[:, 0]
    width = boxes[:, 3] - boxes[:, 1]
    center_y = boxes[:, 0] + 0.5 * height
    center_x = boxes[:, 1] + 0.5 * width
    # Apply deltas
    center_y += deltas[:, 0] * height
    center_x += deltas[:, 1] * width
    height *= tf.exp(deltas[:, 2])
    width *= tf.exp(deltas[:, 3])
    # Convert back to y1, x1, y2, x2
    y1 = center_y - 0.5 * height
    x1 = center_x - 0.5 * width
    y2 = y1 + height
```

```
    x2 = x1 + width
    result = tf.stack([y1, x1, y2, x2], axis=1, name="apply_box_deltas_out")
    return result


def clip_boxes_graph(boxes, window):
    """
    boxes: [N, (y1, x1, y2, x2)]
    window: [4] in the form y1, x1, y2, x2
    """
    # Split
    wy1, wx1, wy2, wx2 = tf.split(window, 4)
    y1, x1, y2, x2 = tf.split(boxes, 4, axis=1)
    # Clip
    y1 = tf.maximum(tf.minimum(y1, wy2), wy1)
    x1 = tf.maximum(tf.minimum(x1, wx2), wx1)
    y2 = tf.maximum(tf.minimum(y2, wy2), wy1)
    x2 = tf.maximum(tf.minimum(x2, wx2), wx1)
    clipped = tf.concat([y1, x1, y2, x2], axis=1, name="clipped_boxes")
    clipped.set_shape((clipped.shape[0], 4))
    return clipped


class ProposalLayer(KE.Layer):
    """Receives anchor scores and selects a subset to pass as proposals
    to the second stage. Filtering is done based on anchor scores and
    non-max suppression to remove overlaps. It also applies bounding
    box refinement deltas to anchors.

    Inputs:
        rpn_probs: [batch, num_anchors, (bg prob, fg prob)]
        rpn_bbox: [batch, num_anchors, (dy, dx, log(dh), log(dw))]
        anchors: [batch, num_anchors, (y1, x1, y2, x2)] anchors in normalized coordinates

    Returns:
        Proposals in normalized coordinates [batch, rois, (y1, x1, y2, x2)]
    """

    def __init__(self, proposal_count, nms_threshold, config=None, **kwargs):
        super(ProposalLayer, self).__init__(**kwargs)
        self.config = config
        self.proposal_count = proposal_count
        self.nms_threshold = nms_threshold

    def call(self, inputs):
        # Box Scores. Use the foreground class confidence. [Batch, num_rois, 1]
        scores = inputs[0][:, :, 1]
        # Box deltas [batch, num_rois, 4]
```

```
deltas = inputs[1]
deltas = deltas * np.reshape(self.config.RPN_BBOX_STD_DEV, [1, 1, 4])
# Anchors
anchors = inputs[2]

# Improve performance by trimming to top anchors by score
# and doing the rest on the smaller subset.
pre_nms_limit = tf.minimum(self.config.PRE_NMS_LIMIT, tf.shape(anchors)[1])
ix = tf.nn.top_k(scores, pre_nms_limit, sorted=True,
                 name="top_anchors").indices
scores = utils.batch_slice([scores, ix], lambda x, y: tf.gather(x, y),
                           self.config.IMAGES_PER_GPU)
deltas = utils.batch_slice([deltas, ix], lambda x, y: tf.gather(x, y),
                           self.config.IMAGES_PER_GPU)
pre_nms_anchors = utils.batch_slice([anchors, ix], lambda a, x: tf.gather(a, x),
                            self.config.IMAGES_PER_GPU,
                            names=["pre_nms_anchors"])

# Apply deltas to anchors to get refined anchors.
# [batch, N, (y1, x1, y2, x2)]
boxes = utils.batch_slice([pre_nms_anchors, deltas],
                          lambda x, y: apply_box_deltas_graph(x, y),
                          self.config.IMAGES_PER_GPU,
                          names=["refined_anchors"])

# Clip to image boundaries. Since we're in normalized coordinates,
# clip to 0..1 range. [batch, N, (y1, x1, y2, x2)]
window = np.array([0, 0, 1, 1], dtype=np.float32)
boxes = utils.batch_slice(boxes,
                          lambda x: clip_boxes_graph(x, window),
                          self.config.IMAGES_PER_GPU,
                          names=["refined_anchors_clipped"])

# Filter out small boxes
# According to Xinlei Chen's paper, this reduces detection accuracy
# for small objects, so we're skipping it.

# Non-max suppression
def nms(boxes, scores):
    indices = tf.image.non_max_suppression(
        boxes, scores, self.proposal_count,
        self.nms_threshold, name="rpn_non_max_suppression")
    proposals = tf.gather(boxes, indices)
    # Pad if needed
    padding = tf.maximum(self.proposal_count - tf.shape(proposals)[0], 0)
    proposals = tf.pad(proposals, [(0, padding), (0, 0)])
    return proposals
proposals = utils.batch_slice([boxes, scores], nms,
```

```
                                    self.config.IMAGES_PER_GPU)
        return proposals

    def compute_output_shape(self, input_shape):
        return (None, self.proposal_count, 4)


############################################################
#  ROIAlign Layer
############################################################

def log2_graph(x):
    """Implementation of Log2. TF doesn't have a native implementation."""
    return tf.log(x) / tf.log(2.0)


class PyramidROIAlign(KE.Layer):
    """Implements ROI Pooling on multiple levels of the feature pyramid.

    Params:
    - pool_shape: [pool_height, pool_width] of the output pooled regions. Usually [7, 7]

    Inputs:
    - boxes: [batch, num_boxes, (y1, x1, y2, x2)] in normalized
             coordinates. Possibly padded with zeros if not enough
             boxes to fill the array.
    - image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    - feature_maps: List of feature maps from different levels of the pyramid.
                    Each is [batch, height, width, channels]

    Output:
    Pooled regions in the shape: [batch, num_boxes, pool_height, pool_width, channels].
    The width and height are those specific in the pool_shape in the layer
    constructor.
    """

    def __init__(self, pool_shape, **kwargs):
        super(PyramidROIAlign, self).__init__(**kwargs)
        self.pool_shape = tuple(pool_shape)

    def call(self, inputs):
        # Crop boxes [batch, num_boxes, (y1, x1, y2, x2)] in normalized coords
        boxes = inputs[0]

        # Image meta
        # Holds details about the image. See compose_image_meta()
        image_meta = inputs[1]
```

```
# Feature Maps. List of feature maps from different level of the
# feature pyramid. Each is [batch, height, width, channels]
feature_maps = inputs[2:]

# Assign each ROI to a level in the pyramid based on the ROI area.
y1, x1, y2, x2 = tf.split(boxes, 4, axis=2)
h = y2 - y1
w = x2 - x1
# Use shape of first image. Images in a batch must have the same size.
image_shape = parse_image_meta_graph(image_meta)['image_shape'][0]
# Equation 1 in the Feature Pyramid Networks paper. Account for
# the fact that our coordinates are normalized here.
# e.g. a 224x224 ROI (in pixels) maps to P4
image_area = tf.cast(image_shape[0] * image_shape[1], tf.float32)
roi_level = log2_graph(tf.sqrt(h * w) / (224.0 / tf.sqrt(image_area)))
roi_level = tf.minimum(5, tf.maximum(
    2, 4 + tf.cast(tf.round(roi_level), tf.int32)))
roi_level = tf.squeeze(roi_level, 2)

# Loop through levels and apply ROI pooling to each. P2 to P5.
pooled = []
box_to_level = []
for i, level in enumerate(range(2, 6)):
    ix = tf.where(tf.equal(roi_level, level))
    level_boxes = tf.gather_nd(boxes, ix)

    # Box indices for crop_and_resize.
    box_indices = tf.cast(ix[:, 0], tf.int32)

    # Keep track of which box is mapped to which level
    box_to_level.append(ix)

    # Stop gradient propogation to ROI proposals
    level_boxes = tf.stop_gradient(level_boxes)
    box_indices = tf.stop_gradient(box_indices)

    # Crop and Resize
    # From Mask R-CNN paper: "We sample four regular locations, so
    # that we can evaluate either max or average pooling. In fact,
    # interpolating only a single value at each bin center (without
    # pooling) is nearly as effective."
    #
    # Here we use the simplified approach of a single value per bin,
    # which is how it's done in tf.crop_and_resize()
    # Result: [batch * num_boxes, pool_height, pool_width, channels]
    pooled.append(tf.image.crop_and_resize(
        feature_maps[i], level_boxes, box_indices, self.pool_shape,
        method="bilinear"))
```

```
        # Pack pooled features into one tensor
        pooled = tf.concat(pooled, axis=0)

        # Pack box_to_level mapping into one array and add another
        # column representing the order of pooled boxes
        box_to_level = tf.concat(box_to_level, axis=0)
        box_range = tf.expand_dims(tf.range(tf.shape(box_to_level)[0]), 1)
        box_to_level = tf.concat([tf.cast(box_to_level, tf.int32), box_range],
                                 axis=1)

        # Rearrange pooled features to match the order of the original boxes
        # Sort box_to_level by batch then box index
        # TF doesn't have a way to sort by two columns, so merge them and sort.
        sorting_tensor = box_to_level[:, 0] * 100000 + box_to_level[:, 1]
        ix = tf.nn.top_k(sorting_tensor, k=tf.shape(
            box_to_level)[0]).indices[::-1]
        ix = tf.gather(box_to_level[:, 2], ix)
        pooled = tf.gather(pooled, ix)

        # Re-add the batch dimension
        shape = tf.concat([tf.shape(boxes)[:2], tf.shape(pooled)[1:]], axis=0)
        pooled = tf.reshape(pooled, shape)
        return pooled

    def compute_output_shape(self, input_shape):
        return input_shape[0][:2] + self.pool_shape + (input_shape[2][-1], )


############################################################
#  Detection Target Layer
############################################################

def overlaps_graph(boxes1, boxes2):
    """Computes IoU overlaps between two sets of boxes.
    boxes1, boxes2: [N, (y1, x1, y2, x2)].
    """
    # 1. Tile boxes2 and repeat boxes1. This allows us to compare
    # every boxes1 against every boxes2 without loops.
    # TF doesn't have an equivalent to np.repeat() so simulate it
    # using tf.tile() and tf.reshape.
    b1 = tf.reshape(tf.tile(tf.expand_dims(boxes1, 1),
                            [1, 1, tf.shape(boxes2)[0]]), [-1, 4])
    b2 = tf.tile(boxes2, [tf.shape(boxes1)[0], 1])
    # 2. Compute intersections
    b1_y1, b1_x1, b1_y2, b1_x2 = tf.split(b1, 4, axis=1)
    b2_y1, b2_x1, b2_y2, b2_x2 = tf.split(b2, 4, axis=1)
    y1 = tf.maximum(b1_y1, b2_y1)
```

```
    x1 = tf.maximum(b1_x1, b2_x1)
    y2 = tf.minimum(b1_y2, b2_y2)
    x2 = tf.minimum(b1_x2, b2_x2)
    intersection = tf.maximum(x2 - x1, 0) * tf.maximum(y2 - y1, 0)
    # 3. Compute unions
    b1_area = (b1_y2 - b1_y1) * (b1_x2 - b1_x1)
    b2_area = (b2_y2 - b2_y1) * (b2_x2 - b2_x1)
    union = b1_area + b2_area - intersection
    # 4. Compute IoU and reshape to [boxes1, boxes2]
    iou = intersection / union
    overlaps = tf.reshape(iou, [tf.shape(boxes1)[0], tf.shape(boxes2)[0]])
    return overlaps


def detection_targets_graph(proposals, gt_class_ids, gt_boxes, gt_masks, config):
    """Generates detection targets for one image. Subsamples proposals and
    generates target class IDs, bounding box deltas, and masks for each.

    Inputs:
    proposals: [POST_NMS_ROIS_TRAINING, (y1, x1, y2, x2)] in normalized coordinates. Might
               be zero padded if there are not enough proposals.
    gt_class_ids: [MAX_GT_INSTANCES] int class IDs
    gt_boxes: [MAX_GT_INSTANCES, (y1, x1, y2, x2)] in normalized coordinates.
    gt_masks: [height, width, MAX_GT_INSTANCES] of boolean type.

    Returns: Target ROIs and corresponding class IDs, bounding box shifts,
    and masks.
    rois: [TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)] in normalized coordinates
    class_ids: [TRAIN_ROIS_PER_IMAGE]. Integer class IDs. Zero padded.
    deltas: [TRAIN_ROIS_PER_IMAGE, (dy, dx, log(dh), log(dw))]
    masks: [TRAIN_ROIS_PER_IMAGE, height, width]. Masks cropped to bbox
           boundaries and resized to neural network output size.

    Note: Returned arrays might be zero padded if not enough target ROIs.
    """
    # Assertions
    asserts = [
        tf.Assert(tf.greater(tf.shape(proposals)[0], 0), [proposals],
                  name="roi_assertion"),
    ]
    with tf.control_dependencies(asserts):
        proposals = tf.identity(proposals)

    # Remove zero padding
    proposals, _ = trim_zeros_graph(proposals, name="trim_proposals")
    gt_boxes, non_zeros = trim_zeros_graph(gt_boxes, name="trim_gt_boxes")
    gt_class_ids = tf.boolean_mask(gt_class_ids, non_zeros,
                                   name="trim_gt_class_ids")
```

```
gt_masks = tf.gather(gt_masks, tf.where(non_zeros)[:, 0], axis=2,
                     name="trim_gt_masks")

# Handle COCO crowds
# A crowd box in COCO is a bounding box around several instances. Exclude
# them from training. A crowd box is given a negative class ID.
crowd_ix = tf.where(gt_class_ids < 0)[:, 0]
non_crowd_ix = tf.where(gt_class_ids > 0)[:, 0]
crowd_boxes = tf.gather(gt_boxes, crowd_ix)
gt_class_ids = tf.gather(gt_class_ids, non_crowd_ix)
gt_boxes = tf.gather(gt_boxes, non_crowd_ix)
gt_masks = tf.gather(gt_masks, non_crowd_ix, axis=2)

# Compute overlaps matrix [proposals, gt_boxes]
overlaps = overlaps_graph(proposals, gt_boxes)

# Compute overlaps with crowd boxes [proposals, crowd_boxes]
crowd_overlaps = overlaps_graph(proposals, crowd_boxes)
crowd_iou_max = tf.reduce_max(crowd_overlaps, axis=1)
no_crowd_bool = (crowd_iou_max < 0.001)

# Determine positive and negative ROIs
roi_iou_max = tf.reduce_max(overlaps, axis=1)
# 1. Positive ROIs are those with >= 0.5 IoU with a GT box
positive_roi_bool = (roi_iou_max >= 0.5)
positive_indices = tf.where(positive_roi_bool)[:, 0]
# 2. Negative ROIs are those with < 0.5 with every GT box. Skip crowds.
negative_indices = tf.where(tf.logical_and(roi_iou_max < 0.5, no_crowd_bool))[:, 0]

# Subsample ROIs. Aim for 33% positive
# Positive ROIs
positive_count = int(config.TRAIN_ROIS_PER_IMAGE *
                     config.ROI_POSITIVE_RATIO)
positive_indices = tf.random_shuffle(positive_indices)[:positive_count]
positive_count = tf.shape(positive_indices)[0]
# Negative ROIs. Add enough to maintain positive:negative ratio.
r = 1.0 / config.ROI_POSITIVE_RATIO
negative_count = tf.cast(r * tf.cast(positive_count, tf.float32), tf.int32) - positive_count
negative_indices = tf.random_shuffle(negative_indices)[:negative_count]
# Gather selected ROIs
positive_rois = tf.gather(proposals, positive_indices)
negative_rois = tf.gather(proposals, negative_indices)

# Assign positive ROIs to GT boxes.
positive_overlaps = tf.gather(overlaps, positive_indices)
roi_gt_box_assignment = tf.cond(
    tf.greater(tf.shape(positive_overlaps)[1], 0),
    true_fn = lambda: tf.argmax(positive_overlaps, axis=1),
```

```
        false_fn = lambda: tf.cast(tf.constant([]),tf.int64)
    )
    roi_gt_boxes = tf.gather(gt_boxes, roi_gt_box_assignment)
    roi_gt_class_ids = tf.gather(gt_class_ids, roi_gt_box_assignment)

    # Compute bbox refinement for positive ROIs
    deltas = utils.box_refinement_graph(positive_rois, roi_gt_boxes)
    deltas /= config.BBOX_STD_DEV

    # Assign positive ROIs to GT masks
    # Permute masks to [N, height, width, 1]
    transposed_masks = tf.expand_dims(tf.transpose(gt_masks, [2, 0, 1]), -1)
    # Pick the right mask for each ROI
    roi_masks = tf.gather(transposed_masks, roi_gt_box_assignment)

    # Compute mask targets
    boxes = positive_rois
    if config.USE_MINI_MASK:
        # Transform ROI coordinates from normalized image space
        # to normalized mini-mask space.
        y1, x1, y2, x2 = tf.split(positive_rois, 4, axis=1)
        gt_y1, gt_x1, gt_y2, gt_x2 = tf.split(roi_gt_boxes, 4, axis=1)
        gt_h = gt_y2 - gt_y1
        gt_w = gt_x2 - gt_x1
        y1 = (y1 - gt_y1) / gt_h
        x1 = (x1 - gt_x1) / gt_w
        y2 = (y2 - gt_y1) / gt_h
        x2 = (x2 - gt_x1) / gt_w
        boxes = tf.concat([y1, x1, y2, x2], 1)
    box_ids = tf.range(0, tf.shape(roi_masks)[0])
    masks = tf.image.crop_and_resize(tf.cast(roi_masks, tf.float32), boxes,
                                     box_ids,
                                     config.MASK_SHAPE)
    # Remove the extra dimension from masks.
    masks = tf.squeeze(masks, axis=3)

    # Threshold mask pixels at 0.5 to have GT masks be 0 or 1 to use with
    # binary cross entropy loss.
    masks = tf.round(masks)

    # Append negative ROIs and pad bbox deltas and masks that
    # are not used for negative ROIs with zeros.
    rois = tf.concat([positive_rois, negative_rois], axis=0)
    N = tf.shape(negative_rois)[0]
    P = tf.maximum(config.TRAIN_ROIS_PER_IMAGE - tf.shape(rois)[0], 0)
    rois = tf.pad(rois, [(0, P), (0, 0)])
    roi_gt_boxes = tf.pad(roi_gt_boxes, [(0, N + P), (0, 0)])
    roi_gt_class_ids = tf.pad(roi_gt_class_ids, [(0, N + P)])
```

```
    deltas = tf.pad(deltas, [(0, N + P), (0, 0)])
    masks = tf.pad(masks, [[0, N + P], (0, 0), (0, 0)])

    return rois, roi_gt_class_ids, deltas, masks


class DetectionTargetLayer(KE.Layer):
    """Subsamples proposals and generates target box refinement, class_ids,
    and masks for each.

    Inputs:
    proposals: [batch, N, (y1, x1, y2, x2)] in normalized coordinates. Might
               be zero padded if there are not enough proposals.
    gt_class_ids: [batch, MAX_GT_INSTANCES] Integer class IDs.
    gt_boxes: [batch, MAX_GT_INSTANCES, (y1, x1, y2, x2)] in normalized
              coordinates.
    gt_masks: [batch, height, width, MAX_GT_INSTANCES] of boolean type

    Returns: Target ROIs and corresponding class IDs, bounding box shifts,
    and masks.
    rois: [batch, TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)] in normalized
          coordinates
    target_class_ids: [batch, TRAIN_ROIS_PER_IMAGE]. Integer class IDs.
    target_deltas: [batch, TRAIN_ROIS_PER_IMAGE, (dy, dx, log(dh), log(dw)]
    target_mask: [batch, TRAIN_ROIS_PER_IMAGE, height, width]
                 Masks cropped to bbox boundaries and resized to neural
                 network output size.

    Note: Returned arrays might be zero padded if not enough target ROIs.
    """

    def __init__(self, config, **kwargs):
        super(DetectionTargetLayer, self).__init__(**kwargs)
        self.config = config

    def call(self, inputs):
        proposals = inputs[0]
        gt_class_ids = inputs[1]
        gt_boxes = inputs[2]
        gt_masks = inputs[3]

        # Slice the batch and run a graph for each slice
        # TODO: Rename target_bbox to target_deltas for clarity
        names = ["rois", "target_class_ids", "target_bbox", "target_mask"]
        outputs = utils.batch_slice(
            [proposals, gt_class_ids, gt_boxes, gt_masks],
            lambda w, x, y, z: detection_targets_graph(
                w, x, y, z, self.config),
```

```
            self.config.IMAGES_PER_GPU, names=names)
        return outputs

    def compute_output_shape(self, input_shape):
        return [
            (None, self.config.TRAIN_ROIS_PER_IMAGE, 4),  # rois
            (None, self.config.TRAIN_ROIS_PER_IMAGE),  # class_ids
            (None, self.config.TRAIN_ROIS_PER_IMAGE, 4),  # deltas
            (None, self.config.TRAIN_ROIS_PER_IMAGE, self.config.MASK_SHAPE[0],
             self.config.MASK_SHAPE[1])  # masks
        ]

    def compute_mask(self, inputs, mask=None):
        return [None, None, None, None]


############################################################
#  Detection Layer
############################################################

def refine_detections_graph(rois, probs, deltas, window, config):
    """Refine classified proposals and filter overlaps and return final
    detections.

    Inputs:
        rois: [N, (y1, x1, y2, x2)] in normalized coordinates
        probs: [N, num_classes]. Class probabilities.
        deltas: [N, num_classes, (dy, dx, log(dh), log(dw))]. Class-specific
                bounding box deltas.
        window: (y1, x1, y2, x2) in normalized coordinates. The part of the image
            that contains the image excluding the padding.

    Returns detections shaped: [num_detections, (y1, x1, y2, x2, class_id, score)] where
        coordinates are normalized.
    """
    # Class IDs per ROI
    class_ids = tf.argmax(probs, axis=1, output_type=tf.int32)
    # Class probability of the top class of each ROI
    indices = tf.stack([tf.range(probs.shape[0]), class_ids], axis=1)
    class_scores = tf.gather_nd(probs, indices)
    # Class-specific bounding box deltas
    deltas_specific = tf.gather_nd(deltas, indices)
    # Apply bounding box deltas
    # Shape: [boxes, (y1, x1, y2, x2)] in normalized coordinates
    refined_rois = apply_box_deltas_graph(
        rois, deltas_specific * config.BBOX_STD_DEV)
    # Clip boxes to image window
    refined_rois = clip_boxes_graph(refined_rois, window)
```

```
# TODO: Filter out boxes with zero area

# Filter out background boxes
keep = tf.where(class_ids > 0)[:, 0]
# Filter out low confidence boxes
if config.DETECTION_MIN_CONFIDENCE:
    conf_keep = tf.where(class_scores >= config.DETECTION_MIN_CONFIDENCE)[:, 0]
    keep = tf.sets.set_intersection(tf.expand_dims(keep, 0),
                                    tf.expand_dims(conf_keep, 0))
    keep = tf.sparse_tensor_to_dense(keep)[0]

# Apply per-class NMS
# 1. Prepare variables
pre_nms_class_ids = tf.gather(class_ids, keep)
pre_nms_scores = tf.gather(class_scores, keep)
pre_nms_rois = tf.gather(refined_rois,   keep)
unique_pre_nms_class_ids = tf.unique(pre_nms_class_ids)[0]

def nms_keep_map(class_id):
    """Apply Non-Maximum Suppression on ROIs of the given class."""
    # Indices of ROIs of the given class
    ixs = tf.where(tf.equal(pre_nms_class_ids, class_id))[:, 0]
    # Apply NMS
    class_keep = tf.image.non_max_suppression(
            tf.gather(pre_nms_rois, ixs),
            tf.gather(pre_nms_scores, ixs),
            max_output_size=config.DETECTION_MAX_INSTANCES,
            iou_threshold=config.DETECTION_NMS_THRESHOLD)
    # Map indices
    class_keep = tf.gather(keep, tf.gather(ixs, class_keep))
    # Pad with -1 so returned tensors have the same shape
    gap = config.DETECTION_MAX_INSTANCES - tf.shape(class_keep)[0]
    class_keep = tf.pad(class_keep, [(0, gap)],
                        mode='CONSTANT', constant_values=-1)
    # Set shape so map_fn() can infer result shape
    class_keep.set_shape([config.DETECTION_MAX_INSTANCES])
    return class_keep

# 2. Map over class IDs
nms_keep = tf.map_fn(nms_keep_map, unique_pre_nms_class_ids,
                     dtype=tf.int64)
# 3. Merge results into one list, and remove -1 padding
nms_keep = tf.reshape(nms_keep, [-1])
nms_keep = tf.gather(nms_keep, tf.where(nms_keep > -1)[:, 0])
# 4. Compute intersection between keep and nms_keep
keep = tf.sets.set_intersection(tf.expand_dims(keep, 0),
                                tf.expand_dims(nms_keep, 0))
```

```
    keep = tf.sparse_tensor_to_dense(keep)[0]
    # Keep top detections
    roi_count = config.DETECTION_MAX_INSTANCES
    class_scores_keep = tf.gather(class_scores, keep)
    num_keep = tf.minimum(tf.shape(class_scores_keep)[0], roi_count)
    top_ids = tf.nn.top_k(class_scores_keep, k=num_keep, sorted=True)[1]
    keep = tf.gather(keep, top_ids)

    # Arrange output as [N, (y1, x1, y2, x2, class_id, score)]
    # Coordinates are normalized.
    detections = tf.concat([
        tf.gather(refined_rois, keep),
        tf.to_float(tf.gather(class_ids, keep))[..., tf.newaxis],
        tf.gather(class_scores, keep)[..., tf.newaxis]
        ], axis=1)

    # Pad with zeros if detections < DETECTION_MAX_INSTANCES
    gap = config.DETECTION_MAX_INSTANCES - tf.shape(detections)[0]
    detections = tf.pad(detections, [(0, gap), (0, 0)], "CONSTANT")
    return detections


class DetectionLayer(KE.Layer):
    """Takes classified proposal boxes and their bounding box deltas and
    returns the final detection boxes.

    Returns:
    [batch, num_detections, (y1, x1, y2, x2, class_id, class_score)] where
    coordinates are normalized.
    """

    def __init__(self, config=None, **kwargs):
        super(DetectionLayer, self).__init__(**kwargs)
        self.config = config

    def call(self, inputs):
        rois = inputs[0]
        mrcnn_class = inputs[1]
        mrcnn_bbox = inputs[2]
        image_meta = inputs[3]

        # Get windows of images in normalized coordinates. Windows are the area
        # in the image that excludes the padding.
        # Use the shape of the first image in the batch to normalize the window
        # because we know that all images get resized to the same size.
        m = parse_image_meta_graph(image_meta)
        image_shape = m['image_shape'][0]
        window = norm_boxes_graph(m['window'], image_shape[:2])
```

```
        # Run detection refinement graph on each item in the batch
        detections_batch = utils.batch_slice(
            [rois, mrcnn_class, mrcnn_bbox, window],
            lambda x, y, w, z: refine_detections_graph(x, y, w, z, self.config),
            self.config.IMAGES_PER_GPU)

        # Reshape output
        # [batch, num_detections, (y1, x1, y2, x2, class_id, class_score)] in
        # normalized coordinates
        return tf.reshape(
            detections_batch,
            [self.config.BATCH_SIZE, self.config.DETECTION_MAX_INSTANCES, 6])

    def compute_output_shape(self, input_shape):
        return (None, self.config.DETECTION_MAX_INSTANCES, 6)


############################################################
#  Region Proposal Network (RPN)
############################################################

def rpn_graph(feature_map, anchors_per_location, anchor_stride):
    """Builds the computation graph of Region Proposal Network.

    feature_map: backbone features [batch, height, width, depth]
    anchors_per_location: number of anchors per pixel in the feature map
    anchor_stride: Controls the density of anchors. Typically 1 (anchors for
                   every pixel in the feature map), or 2 (every other pixel).

    Returns:
        rpn_class_logits: [batch, H * W * anchors_per_location, 2] Anchor classifier logits (before sof
        rpn_probs: [batch, H * W * anchors_per_location, 2] Anchor classifier probabilities.
        rpn_bbox: [batch, H * W * anchors_per_location, (dy, dx, log(dh), log(dw))] Deltas to be
                  applied to anchors.
    """
    # TODO: check if stride of 2 causes alignment issues if the feature map
    # is not even.
    # Shared convolutional base of the RPN
    shared = KL.Conv2D(512, (3, 3), padding='same', activation='relu',
                       strides=anchor_stride,
                       name='rpn_conv_shared')(feature_map)

    # Anchor Score. [batch, height, width, anchors per location * 2].
    x = KL.Conv2D(2 * anchors_per_location, (1, 1), padding='valid',
                  activation='linear', name='rpn_class_raw')(shared)

    # Reshape to [batch, anchors, 2]
```

```
    rpn_class_logits = KL.Lambda(
        lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 2]))(x)

    # Softmax on last dimension of BG/FG.
    rpn_probs = KL.Activation(
        "softmax", name="rpn_class_xxx")(rpn_class_logits)

    # Bounding box refinement. [batch, H, W, anchors per location * depth]
    # where depth is [x, y, log(w), log(h)]
    x = KL.Conv2D(anchors_per_location * 4, (1, 1), padding="valid",
                  activation='linear', name='rpn_bbox_pred')(shared)

    # Reshape to [batch, anchors, 4]
    rpn_bbox = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 4]))(x)

    return [rpn_class_logits, rpn_probs, rpn_bbox]


def build_rpn_model(anchor_stride, anchors_per_location, depth):
    """Builds a Keras model of the Region Proposal Network.
    It wraps the RPN graph so it can be used multiple times with shared
    weights.

    anchors_per_location: number of anchors per pixel in the feature map
    anchor_stride: Controls the density of anchors. Typically 1 (anchors for
                   every pixel in the feature map), or 2 (every other pixel).
    depth: Depth of the backbone feature map.

    Returns a Keras Model object. The model outputs, when called, are:
    rpn_class_logits: [batch, H * W * anchors_per_location, 2] Anchor classifier logits (before softmax
    rpn_probs: [batch, H * W * anchors_per_location, 2] Anchor classifier probabilities.
    rpn_bbox: [batch, H * W * anchors_per_location, (dy, dx, log(dh), log(dw))] Deltas to be
                applied to anchors.
    """
    input_feature_map = KL.Input(shape=[None, None, depth],
                                 name="input_rpn_feature_map")
    outputs = rpn_graph(input_feature_map, anchors_per_location, anchor_stride)
    return KM.Model([input_feature_map], outputs, name="rpn_model")


############################################################
#  Feature Pyramid Network Heads
############################################################

def fpn_classifier_graph(rois, feature_maps, image_meta,
                         pool_size, num_classes, train_bn=True,
                         fc_layers_size=1024):
    """Builds the computation graph of the feature pyramid network classifier
```

```
    and regressor heads.

    rois: [batch, num_rois, (y1, x1, y2, x2)] Proposal boxes in normalized
          coordinates.
    feature_maps: List of feature maps from different layers of the pyramid,
                  [P2, P3, P4, P5]. Each has a different resolution.
    image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    pool_size: The width of the square feature map generated from ROI Pooling.
    num_classes: number of classes, which determines the depth of the results
    train_bn: Boolean. Train or freeze Batch Norm layers
    fc_layers_size: Size of the 2 FC layers

    Returns:
        logits: [batch, num_rois, NUM_CLASSES] classifier logits (before softmax)
        probs: [batch, num_rois, NUM_CLASSES] classifier probabilities
        bbox_deltas: [batch, num_rois, NUM_CLASSES, (dy, dx, log(dh), log(dw))] Deltas to apply to
                     proposal boxes
    """
    # ROI Pooling
    # Shape: [batch, num_rois, POOL_SIZE, POOL_SIZE, channels]
    x = PyramidROIAlign([pool_size, pool_size],
                        name="roi_align_classifier")([rois, image_meta] + feature_maps)
    # Two 1024 FC layers (implemented with Conv2D for consistency)
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (pool_size, pool_size), padding="valid"),
                           name="mrcnn_class_conv1")(x)
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn1')(x, training=train_bn)
    x = KL.Activation('relu')(x)
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (1, 1)),
                           name="mrcnn_class_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn2')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    shared = KL.Lambda(lambda x: K.squeeze(K.squeeze(x, 3), 2),
                       name="pool_squeeze")(x)

    # Classifier head
    mrcnn_class_logits = KL.TimeDistributed(KL.Dense(num_classes),
                                            name='mrcnn_class_logits')(shared)
    mrcnn_probs = KL.TimeDistributed(KL.Activation("softmax"),
                                     name="mrcnn_class")(mrcnn_class_logits)

    # BBox head
    # [batch, num_rois, NUM_CLASSES * (dy, dx, log(dh), log(dw))]
    x = KL.TimeDistributed(KL.Dense(num_classes * 4, activation='linear'),
                           name='mrcnn_bbox_fc')(shared)
    # Reshape to [batch, num_rois, NUM_CLASSES, (dy, dx, log(dh), log(dw))]
    s = K.int_shape(x)
    mrcnn_bbox = KL.Reshape((s[1], num_classes, 4), name="mrcnn_bbox")(x)
```

```
    return mrcnn_class_logits, mrcnn_probs, mrcnn_bbox


def build_fpn_mask_graph(rois, feature_maps, image_meta,
                         pool_size, num_classes, train_bn=True):
    """Builds the computation graph of the mask head of Feature Pyramid Network.

    rois: [batch, num_rois, (y1, x1, y2, x2)] Proposal boxes in normalized
          coordinates.
    feature_maps: List of feature maps from different layers of the pyramid,
                  [P2, P3, P4, P5]. Each has a different resolution.
    image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    pool_size: The width of the square feature map generated from ROI Pooling.
    num_classes: number of classes, which determines the depth of the results
    train_bn: Boolean. Train or freeze Batch Norm layers

    Returns: Masks [batch, num_rois, MASK_POOL_SIZE, MASK_POOL_SIZE, NUM_CLASSES]
    """
    # ROI Pooling
    # Shape: [batch, num_rois, MASK_POOL_SIZE, MASK_POOL_SIZE, channels]
    x = PyramidROIAlign([pool_size, pool_size],
                        name="roi_align_mask")([rois, image_meta] + feature_maps)

    # Conv layers
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv1")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn1')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn2')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv3")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn3')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv4")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn4')(x, training=train_bn)
    x = KL.Activation('relu')(x)
```

```
    x = KL.TimeDistributed(KL.Conv2DTranspose(256, (2, 2), strides=2, activation="relu"),
                           name="mrcnn_mask_deconv")(x)
    x = KL.TimeDistributed(KL.Conv2D(num_classes, (1, 1), strides=1, activation="sigmoid"),
                           name="mrcnn_mask")(x)
    return x


############################################################
#  Loss Functions
############################################################

def smooth_l1_loss(y_true, y_pred):
    """Implements Smooth-L1 loss.
    y_true and y_pred are typically: [N, 4], but could be any shape.
    """
    diff = K.abs(y_true - y_pred)
    less_than_one = K.cast(K.less(diff, 1.0), "float32")
    loss = (less_than_one * 0.5 * diff**2) + (1 - less_than_one) * (diff - 0.5)
    return loss


def rpn_class_loss_graph(rpn_match, rpn_class_logits):
    """RPN anchor classifier loss.

    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
               -1=negative, 0=neutral anchor.
    rpn_class_logits: [batch, anchors, 2]. RPN classifier logits for BG/FG.
    """
    # Squeeze last dim to simplify
    rpn_match = tf.squeeze(rpn_match, -1)
    # Get anchor classes. Convert the -1/+1 match to 0/1 values.
    anchor_class = K.cast(K.equal(rpn_match, 1), tf.int32)
    # Positive and Negative anchors contribute to the loss,
    # but neutral anchors (match value = 0) don't.
    indices = tf.where(K.not_equal(rpn_match, 0))
    # Pick rows that contribute to the loss and filter out the rest.
    rpn_class_logits = tf.gather_nd(rpn_class_logits, indices)
    anchor_class = tf.gather_nd(anchor_class, indices)
    # Cross entropy loss
    loss = K.sparse_categorical_crossentropy(target=anchor_class,
                                             output=rpn_class_logits,
                                             from_logits=True)
    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss


def rpn_bbox_loss_graph(config, target_bbox, rpn_match, rpn_bbox):
```

```
    """Return the RPN bounding box loss graph.

    config: the model config object.
    target_bbox: [batch, max positive anchors, (dy, dx, log(dh), log(dw))].
        Uses 0 padding to fill in unsed bbox deltas.
    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
               -1=negative, 0=neutral anchor.
    rpn_bbox: [batch, anchors, (dy, dx, log(dh), log(dw))]
    """
    # Positive anchors contribute to the loss, but negative and
    # neutral anchors (match value of 0 or -1) don't.
    rpn_match = K.squeeze(rpn_match, -1)
    indices = tf.where(K.equal(rpn_match, 1))

    # Pick bbox deltas that contribute to the loss
    rpn_bbox = tf.gather_nd(rpn_bbox, indices)

    # Trim target bounding box deltas to the same length as rpn_bbox.
    batch_counts = K.sum(K.cast(K.equal(rpn_match, 1), tf.int32), axis=1)
    target_bbox = batch_pack_graph(target_bbox, batch_counts,
                                   config.IMAGES_PER_GPU)

    loss = smooth_l1_loss(target_bbox, rpn_bbox)

    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss


def mrcnn_class_loss_graph(target_class_ids, pred_class_logits,
                           active_class_ids):
    """Loss for the classifier head of Mask RCNN.

    target_class_ids: [batch, num_rois]. Integer class IDs. Uses zero
        padding to fill in the array.
    pred_class_logits: [batch, num_rois, num_classes]
    active_class_ids: [batch, num_classes]. Has a value of 1 for
        classes that are in the dataset of the image, and 0
        for classes that are not in the dataset.
    """
    # During model building, Keras calls this function with
    # target_class_ids of type float32. Unclear why. Cast it
    # to int to get around it.
    target_class_ids = tf.cast(target_class_ids, 'int64')

    # Find predictions of classes that are not in the dataset.
    pred_class_ids = tf.argmax(pred_class_logits, axis=2)
    # TODO: Update this line to work with batch > 1. Right now it assumes all
    #       images in a batch have the same active_class_ids
```

```
    pred_active = tf.gather(active_class_ids[0], pred_class_ids)

    # Loss
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels=target_class_ids, logits=pred_class_logits)

    # Erase losses of predictions of classes that are not in the active
    # classes of the image.
    loss = loss * pred_active

    # Computer loss mean. Use only predictions that contribute
    # to the loss to get a correct mean.
    loss = tf.reduce_sum(loss) / tf.reduce_sum(pred_active)
    return loss


def mrcnn_bbox_loss_graph(target_bbox, target_class_ids, pred_bbox):
    """Loss for Mask R-CNN bounding box refinement.

    target_bbox: [batch, num_rois, (dy, dx, log(dh), log(dw))]
    target_class_ids: [batch, num_rois]. Integer class IDs.
    pred_bbox: [batch, num_rois, num_classes, (dy, dx, log(dh), log(dw))]
    """
    # Reshape to merge batch and roi dimensions for simplicity.
    target_class_ids = K.reshape(target_class_ids, (-1,))
    target_bbox = K.reshape(target_bbox, (-1, 4))
    pred_bbox = K.reshape(pred_bbox, (-1, K.int_shape(pred_bbox)[2], 4))

    # Only positive ROIs contribute to the loss. And only
    # the right class_id of each ROI. Get their indices.
    positive_roi_ix = tf.where(target_class_ids > 0)[:, 0]
    positive_roi_class_ids = tf.cast(
        tf.gather(target_class_ids, positive_roi_ix), tf.int64)
    indices = tf.stack([positive_roi_ix, positive_roi_class_ids], axis=1)

    # Gather the deltas (predicted and true) that contribute to loss
    target_bbox = tf.gather(target_bbox, positive_roi_ix)
    pred_bbox = tf.gather_nd(pred_bbox, indices)

    # Smooth-L1 Loss
    loss = K.switch(tf.size(target_bbox) > 0,
                    smooth_l1_loss(y_true=target_bbox, y_pred=pred_bbox),
                    tf.constant(0.0))
    loss = K.mean(loss)
    return loss


def mrcnn_mask_loss_graph(target_masks, target_class_ids, pred_masks):
```

```
    """Mask binary cross-entropy loss for the masks head.

    target_masks: [batch, num_rois, height, width].
        A float32 tensor of values 0 or 1. Uses zero padding to fill array.
    target_class_ids: [batch, num_rois]. Integer class IDs. Zero padded.
    pred_masks: [batch, proposals, height, width, num_classes] float32 tensor
                with values from 0 to 1.
    """
    # Reshape for simplicity. Merge first two dimensions into one.
    target_class_ids = K.reshape(target_class_ids, (-1,))
    mask_shape = tf.shape(target_masks)
    target_masks = K.reshape(target_masks, (-1, mask_shape[2], mask_shape[3]))
    pred_shape = tf.shape(pred_masks)
    pred_masks = K.reshape(pred_masks,
                           (-1, pred_shape[2], pred_shape[3], pred_shape[4]))
    # Permute predicted masks to [N, num_classes, height, width]
    pred_masks = tf.transpose(pred_masks, [0, 3, 1, 2])

    # Only positive ROIs contribute to the loss. And only
    # the class specific mask of each ROI.
    positive_ix = tf.where(target_class_ids > 0)[:, 0]
    positive_class_ids = tf.cast(
        tf.gather(target_class_ids, positive_ix), tf.int64)
    indices = tf.stack([positive_ix, positive_class_ids], axis=1)

    # Gather the masks (predicted and true) that contribute to loss
    y_true = tf.gather(target_masks, positive_ix)
    y_pred = tf.gather_nd(pred_masks, indices)

    # Compute binary cross entropy. If no positive ROIs, then return 0.
    # shape: [batch, roi, num_classes]
    loss = K.switch(tf.size(y_true) > 0,
                    K.binary_crossentropy(target=y_true, output=y_pred),
                    tf.constant(0.0))
    loss = K.mean(loss)
    return loss


############################################################
#  Data Generator
############################################################

def load_image_gt(dataset, config, image_id, augment=False, augmentation=None,
                  use_mini_mask=False):
    """Load and return ground truth data for an image (image, mask, bounding boxes).

    augment: (deprecated. Use augmentation instead). If true, apply random
        image augmentation. Currently, only horizontal flipping is offered.
```

```
    augmentation: Optional. An imgaug (https://github.com/aleju/imgaug) augmentation.
        For example, passing imgaug.augmenters.Fliplr(0.5) flips images
        right/left 50% of the time.
    use_mini_mask: If False, returns full-size masks that are the same height
        and width as the original image. These can be big, for example
        1024x1024x100 (for 100 instances). Mini masks are smaller, typically,
        224x224 and are generated by extracting the bounding box of the
        object and resizing it to MINI_MASK_SHAPE.

    Returns:
    image: [height, width, 3]
    shape: the original shape of the image before resizing and cropping.
    class_ids: [instance_count] Integer class IDs
    bbox: [instance_count, (y1, x1, y2, x2)]
    mask: [height, width, instance_count]. The height and width are those
        of the image unless use_mini_mask is True, in which case they are
        defined in MINI_MASK_SHAPE.
    """
    # Load image and mask
    image = dataset.load_image(image_id)
    mask, class_ids = dataset.load_mask(image_id)
    original_shape = image.shape
    image, window, scale, padding, crop = utils.resize_image(
        image,
        min_dim=config.IMAGE_MIN_DIM,
        min_scale=config.IMAGE_MIN_SCALE,
        max_dim=config.IMAGE_MAX_DIM,
        mode=config.IMAGE_RESIZE_MODE)
    mask = utils.resize_mask(mask, scale, padding, crop)

    # Random horizontal flips.
    # TODO: will be removed in a future update in favor of augmentation
    if augment:
        logging.warning("'augment' is deprecated. Use 'augmentation' instead.")
        if random.randint(0, 1):
            image = np.fliplr(image)
            mask = np.fliplr(mask)

    # Augmentation
    # This requires the imgaug lib (https://github.com/aleju/imgaug)
    if augmentation:
        import imgaug

        # Augmenters that are safe to apply to masks
        # Some, such as Affine, have settings that make them unsafe, so always
        # test your augmentation on masks
        MASK_AUGMENTERS = ["Sequential", "SomeOf", "OneOf", "Sometimes",
                           "Fliplr", "Flipud", "CropAndPad",
```

```
                        "Affine", "PiecewiseAffine"]

        def hook(images, augmenter, parents, default):
            """Determines which augmenters to apply to masks."""
            return augmenter.__class__.__name__ in MASK_AUGMENTERS

        # Store shapes before augmentation to compare
        image_shape = image.shape
        mask_shape = mask.shape
        # Make augmenters deterministic to apply similarly to images and masks
        det = augmentation.to_deterministic()
        image = det.augment_image(image)
        # Change mask to np.uint8 because imgaug doesn't support np.bool
        mask = det.augment_image(mask.astype(np.uint8),
                                 hooks=imgaug.HooksImages(activator=hook))
        # Verify that shapes didn't change
        assert image.shape == image_shape, "Augmentation shouldn't change image size"
        assert mask.shape == mask_shape, "Augmentation shouldn't change mask size"
        # Change mask back to bool
        mask = mask.astype(np.bool)

    # Note that some boxes might be all zeros if the corresponding mask got cropped out.
    # and here is to filter them out
    _idx = np.sum(mask, axis=(0, 1)) > 0
    mask = mask[:, :, _idx]
    class_ids = class_ids[_idx]
    # Bounding boxes. Note that some boxes might be all zeros
    # if the corresponding mask got cropped out.
    # bbox: [num_instances, (y1, x1, y2, x2)]
    bbox = utils.extract_bboxes(mask)

    # Active classes
    # Different datasets have different classes, so track the
    # classes supported in the dataset of this image.
    active_class_ids = np.zeros([dataset.num_classes], dtype=np.int32)
    source_class_ids = dataset.source_class_ids[dataset.image_info[image_id]["source"]]
    active_class_ids[source_class_ids] = 1

    # Resize masks to smaller size to reduce memory usage
    if use_mini_mask:
        mask = utils.minimize_mask(bbox, mask, config.MINI_MASK_SHAPE)

    # Image meta data
    image_meta = compose_image_meta(image_id, original_shape, image.shape,
                                    window, scale, active_class_ids)

    return image, image_meta, class_ids, bbox, mask
```

```
def build_detection_targets(rpn_rois, gt_class_ids, gt_boxes, gt_masks, config):
    """Generate targets for training Stage 2 classifier and mask heads.
    This is not used in normal training. It's useful for debugging or to train
    the Mask RCNN heads without using the RPN head.

    Inputs:
    rpn_rois: [N, (y1, x1, y2, x2)] proposal boxes.
    gt_class_ids: [instance count] Integer class IDs
    gt_boxes: [instance count, (y1, x1, y2, x2)]
    gt_masks: [height, width, instance count] Ground truth masks. Can be full
              size or mini-masks.

    Returns:
    rois: [TRAIN_ROIS_PER_IMAGE, (y1, x1, y2, x2)]
    class_ids: [TRAIN_ROIS_PER_IMAGE]. Integer class IDs.
    bboxes: [TRAIN_ROIS_PER_IMAGE, NUM_CLASSES, (y, x, log(h), log(w))]. Class-specific
            bbox refinements.
    masks: [TRAIN_ROIS_PER_IMAGE, height, width, NUM_CLASSES). Class specific masks cropped
           to bbox boundaries and resized to neural network output size.
    """
    assert rpn_rois.shape[0] > 0
    assert gt_class_ids.dtype == np.int32, "Expected int but got {}".format(
        gt_class_ids.dtype)
    assert gt_boxes.dtype == np.int32, "Expected int but got {}".format(
        gt_boxes.dtype)
    assert gt_masks.dtype == np.bool_, "Expected bool but got {}".format(
        gt_masks.dtype)

    # It's common to add GT Boxes to ROIs but we don't do that here because
    # according to XinLei Chen's paper, it doesn't help.

    # Trim empty padding in gt_boxes and gt_masks parts
    instance_ids = np.where(gt_class_ids > 0)[0]
    assert instance_ids.shape[0] > 0, "Image must contain instances."
    gt_class_ids = gt_class_ids[instance_ids]
    gt_boxes = gt_boxes[instance_ids]
    gt_masks = gt_masks[:, :, instance_ids]

    # Compute areas of ROIs and ground truth boxes.
    rpn_roi_area = (rpn_rois[:, 2] - rpn_rois[:, 0]) * \
        (rpn_rois[:, 3] - rpn_rois[:, 1])
    gt_box_area = (gt_boxes[:, 2] - gt_boxes[:, 0]) * \
        (gt_boxes[:, 3] - gt_boxes[:, 1])

    # Compute overlaps [rpn_rois, gt_boxes]
    overlaps = np.zeros((rpn_rois.shape[0], gt_boxes.shape[0]))
    for i in range(overlaps.shape[1]):
```

```
        gt = gt_boxes[i]
        overlaps[:, i] = utils.compute_iou(
            gt, rpn_rois, gt_box_area[i], rpn_roi_area)

    # Assign ROIs to GT boxes
    rpn_roi_iou_argmax = np.argmax(overlaps, axis=1)
    rpn_roi_iou_max = overlaps[np.arange(
        overlaps.shape[0]), rpn_roi_iou_argmax]
    # GT box assigned to each ROI
    rpn_roi_gt_boxes = gt_boxes[rpn_roi_iou_argmax]
    rpn_roi_gt_class_ids = gt_class_ids[rpn_roi_iou_argmax]

    # Positive ROIs are those with >= 0.5 IoU with a GT box.
    fg_ids = np.where(rpn_roi_iou_max > 0.5)[0]

    # Negative ROIs are those with max IoU 0.1-0.5 (hard example mining)
    # TODO: To hard example mine or not to hard example mine, that's the question
    # bg_ids = np.where((rpn_roi_iou_max >= 0.1) & (rpn_roi_iou_max < 0.5))[0]
    bg_ids = np.where(rpn_roi_iou_max < 0.5)[0]

    # Subsample ROIs. Aim for 33% foreground.
    # FG
    fg_roi_count = int(config.TRAIN_ROIS_PER_IMAGE * config.ROI_POSITIVE_RATIO)
    if fg_ids.shape[0] > fg_roi_count:
        keep_fg_ids = np.random.choice(fg_ids, fg_roi_count, replace=False)
    else:
        keep_fg_ids = fg_ids
    # BG
    remaining = config.TRAIN_ROIS_PER_IMAGE - keep_fg_ids.shape[0]
    if bg_ids.shape[0] > remaining:
        keep_bg_ids = np.random.choice(bg_ids, remaining, replace=False)
    else:
        keep_bg_ids = bg_ids
    # Combine indices of ROIs to keep
    keep = np.concatenate([keep_fg_ids, keep_bg_ids])
    # Need more?
    remaining = config.TRAIN_ROIS_PER_IMAGE - keep.shape[0]
    if remaining > 0:
        # Looks like we don't have enough samples to maintain the desired
        # balance. Reduce requirements and fill in the rest. This is
        # likely different from the Mask RCNN paper.

        # There is a small chance we have neither fg nor bg samples.
        if keep.shape[0] == 0:
            # Pick bg regions with easier IoU threshold
            bg_ids = np.where(rpn_roi_iou_max < 0.5)[0]
            assert bg_ids.shape[0] >= remaining
            keep_bg_ids = np.random.choice(bg_ids, remaining, replace=False)
```

```
            assert keep_bg_ids.shape[0] == remaining
            keep = np.concatenate([keep, keep_bg_ids])
        else:
            # Fill the rest with repeated bg rois.
            keep_extra_ids = np.random.choice(
                keep_bg_ids, remaining, replace=True)
            keep = np.concatenate([keep, keep_extra_ids])
    assert keep.shape[0] == config.TRAIN_ROIS_PER_IMAGE, \
        "keep doesn't match ROI batch size {}, {}".format(
            keep.shape[0], config.TRAIN_ROIS_PER_IMAGE)

    # Reset the gt boxes assigned to BG ROIs.
    rpn_roi_gt_boxes[keep_bg_ids, :] = 0
    rpn_roi_gt_class_ids[keep_bg_ids] = 0

    # For each kept ROI, assign a class_id, and for FG ROIs also add bbox refinement.
    rois = rpn_rois[keep]
    roi_gt_boxes = rpn_roi_gt_boxes[keep]
    roi_gt_class_ids = rpn_roi_gt_class_ids[keep]
    roi_gt_assignment = rpn_roi_iou_argmax[keep]

    # Class-aware bbox deltas. [y, x, log(h), log(w)]
    bboxes = np.zeros((config.TRAIN_ROIS_PER_IMAGE,
                       config.NUM_CLASSES, 4), dtype=np.float32)
    pos_ids = np.where(roi_gt_class_ids > 0)[0]
    bboxes[pos_ids, roi_gt_class_ids[pos_ids]] = utils.box_refinement(
        rois[pos_ids], roi_gt_boxes[pos_ids, :4])
    # Normalize bbox refinements
    bboxes /= config.BBOX_STD_DEV

    # Generate class-specific target masks
    masks = np.zeros((config.TRAIN_ROIS_PER_IMAGE, config.MASK_SHAPE[0], config.MASK_SHAPE[1], config.N
                      dtype=np.float32)
    for i in pos_ids:
        class_id = roi_gt_class_ids[i]
        assert class_id > 0, "class id must be greater than 0"
        gt_id = roi_gt_assignment[i]
        class_mask = gt_masks[:, :, gt_id]

        if config.USE_MINI_MASK:
            # Create a mask placeholder, the size of the image
            placeholder = np.zeros(config.IMAGE_SHAPE[:2], dtype=bool)
            # GT box
            gt_y1, gt_x1, gt_y2, gt_x2 = gt_boxes[gt_id]
            gt_w = gt_x2 - gt_x1
            gt_h = gt_y2 - gt_y1
            # Resize mini mask to size of GT box
            placeholder[gt_y1:gt_y2, gt_x1:gt_x2] = \
```

```
                np.round(utils.resize(class_mask, (gt_h, gt_w))).astype(bool)
            # Place the mini batch in the placeholder
            class_mask = placeholder

        # Pick part of the mask and resize it
        y1, x1, y2, x2 = rois[i].astype(np.int32)
        m = class_mask[y1:y2, x1:x2]
        mask = utils.resize(m, config.MASK_SHAPE)
        masks[i, :, :, class_id] = mask

    return rois, roi_gt_class_ids, bboxes, masks


def build_rpn_targets(image_shape, anchors, gt_class_ids, gt_boxes, config):
    """Given the anchors and GT boxes, compute overlaps and identify positive
    anchors and deltas to refine them to match their corresponding GT boxes.

    anchors: [num_anchors, (y1, x1, y2, x2)]
    gt_class_ids: [num_gt_boxes] Integer class IDs.
    gt_boxes: [num_gt_boxes, (y1, x1, y2, x2)]

    Returns:
    rpn_match: [N] (int32) matches between anchors and GT boxes.
               1 = positive anchor, -1 = negative anchor, 0 = neutral
    rpn_bbox: [N, (dy, dx, log(dh), log(dw))] Anchor bbox deltas.
    """
    # RPN Match: 1 = positive anchor, -1 = negative anchor, 0 = neutral
    rpn_match = np.zeros([anchors.shape[0]], dtype=np.int32)
    # RPN bounding boxes: [max anchors per image, (dy, dx, log(dh), log(dw))]
    rpn_bbox = np.zeros((config.RPN_TRAIN_ANCHORS_PER_IMAGE, 4))

    # Handle COCO crowds
    # A crowd box in COCO is a bounding box around several instances. Exclude
    # them from training. A crowd box is given a negative class ID.
    crowd_ix = np.where(gt_class_ids < 0)[0]
    if crowd_ix.shape[0] > 0:
        # Filter out crowds from ground truth class IDs and boxes
        non_crowd_ix = np.where(gt_class_ids > 0)[0]
        crowd_boxes = gt_boxes[crowd_ix]
        gt_class_ids = gt_class_ids[non_crowd_ix]
        gt_boxes = gt_boxes[non_crowd_ix]
        # Compute overlaps with crowd boxes [anchors, crowds]
        crowd_overlaps = utils.compute_overlaps(anchors, crowd_boxes)
        crowd_iou_max = np.amax(crowd_overlaps, axis=1)
        no_crowd_bool = (crowd_iou_max < 0.001)
    else:
        # All anchors don't intersect a crowd
        no_crowd_bool = np.ones([anchors.shape[0]], dtype=bool)
```

```
# Compute overlaps [num_anchors, num_gt_boxes]
overlaps = utils.compute_overlaps(anchors, gt_boxes)

# Match anchors to GT Boxes
# If an anchor overlaps a GT box with IoU >= 0.7 then it's positive.
# If an anchor overlaps a GT box with IoU < 0.3 then it's negative.
# Neutral anchors are those that don't match the conditions above,
# and they don't influence the loss function.
# However, don't keep any GT box unmatched (rare, but happens). Instead,
# match it to the closest anchor (even if its max IoU is < 0.3).
#
# 1. Set negative anchors first. They get overwritten below if a GT box is
# matched to them. Skip boxes in crowd areas.
anchor_iou_argmax = np.argmax(overlaps, axis=1)
anchor_iou_max = overlaps[np.arange(overlaps.shape[0]), anchor_iou_argmax]
rpn_match[(anchor_iou_max < 0.3) & (no_crowd_bool)] = -1
# 2. Set an anchor for each GT box (regardless of IoU value).
# If multiple anchors have the same IoU match all of them
gt_iou_argmax = np.argwhere(overlaps == np.max(overlaps, axis=0))[:,0]
rpn_match[gt_iou_argmax] = 1
# 3. Set anchors with high overlap as positive.
rpn_match[anchor_iou_max >= 0.7] = 1

# Subsample to balance positive and negative anchors
# Don't let positives be more than half the anchors
ids = np.where(rpn_match == 1)[0]
extra = len(ids) - (config.RPN_TRAIN_ANCHORS_PER_IMAGE // 2)
if extra > 0:
    # Reset the extra ones to neutral
    ids = np.random.choice(ids, extra, replace=False)
    rpn_match[ids] = 0
# Same for negative proposals
ids = np.where(rpn_match == -1)[0]
extra = len(ids) - (config.RPN_TRAIN_ANCHORS_PER_IMAGE -
                    np.sum(rpn_match == 1))
if extra > 0:
    # Rest the extra ones to neutral
    ids = np.random.choice(ids, extra, replace=False)
    rpn_match[ids] = 0

# For positive anchors, compute shift and scale needed to transform them
# to match the corresponding GT boxes.
ids = np.where(rpn_match == 1)[0]
ix = 0  # index into rpn_bbox
# TODO: use box_refinement() rather than duplicating the code here
for i, a in zip(ids, anchors[ids]):
    # Closest gt box (it might have IoU < 0.7)
```

```
            gt = gt_boxes[anchor_iou_argmax[i]]

            # Convert coordinates to center plus width/height.
            # GT Box
            gt_h = gt[2] - gt[0]
            gt_w = gt[3] - gt[1]
            gt_center_y = gt[0] + 0.5 * gt_h
            gt_center_x = gt[1] + 0.5 * gt_w
            # Anchor
            a_h = a[2] - a[0]
            a_w = a[3] - a[1]
            a_center_y = a[0] + 0.5 * a_h
            a_center_x = a[1] + 0.5 * a_w

            # Compute the bbox refinement that the RPN should predict.
            rpn_bbox[ix] = [
                (gt_center_y - a_center_y) / a_h,
                (gt_center_x - a_center_x) / a_w,
                np.log(gt_h / a_h),
                np.log(gt_w / a_w),
            ]
            # Normalize
            rpn_bbox[ix] /= config.RPN_BBOX_STD_DEV
            ix += 1

    return rpn_match, rpn_bbox


def generate_random_rois(image_shape, count, gt_class_ids, gt_boxes):
    """Generates ROI proposals similar to what a region proposal network
    would generate.

    image_shape: [Height, Width, Depth]
    count: Number of ROIs to generate
    gt_class_ids: [N] Integer ground truth class IDs
    gt_boxes: [N, (y1, x1, y2, x2)] Ground truth boxes in pixels.

    Returns: [count, (y1, x1, y2, x2)] ROI boxes in pixels.
    """
    # placeholder
    rois = np.zeros((count, 4), dtype=np.int32)

    # Generate random ROIs around GT boxes (90% of count)
    rois_per_box = int(0.9 * count / gt_boxes.shape[0])
    for i in range(gt_boxes.shape[0]):
        gt_y1, gt_x1, gt_y2, gt_x2 = gt_boxes[i]
        h = gt_y2 - gt_y1
        w = gt_x2 - gt_x1
```

```
        # random boundaries
        r_y1 = max(gt_y1 - h, 0)
        r_y2 = min(gt_y2 + h, image_shape[0])
        r_x1 = max(gt_x1 - w, 0)
        r_x2 = min(gt_x2 + w, image_shape[1])

        # To avoid generating boxes with zero area, we generate double what
        # we need and filter out the extra. If we get fewer valid boxes
        # than we need, we loop and try again.
        while True:
            y1y2 = np.random.randint(r_y1, r_y2, (rois_per_box * 2, 2))
            x1x2 = np.random.randint(r_x1, r_x2, (rois_per_box * 2, 2))
            # Filter out zero area boxes
            threshold = 1
            y1y2 = y1y2[np.abs(y1y2[:, 0] - y1y2[:, 1]) >=
                        threshold][:rois_per_box]
            x1x2 = x1x2[np.abs(x1x2[:, 0] - x1x2[:, 1]) >=
                        threshold][:rois_per_box]
            if y1y2.shape[0] == rois_per_box and x1x2.shape[0] == rois_per_box:
                break

        # Sort on axis 1 to ensure x1 <= x2 and y1 <= y2 and then reshape
        # into x1, y1, x2, y2 order
        x1, x2 = np.split(np.sort(x1x2, axis=1), 2, axis=1)
        y1, y2 = np.split(np.sort(y1y2, axis=1), 2, axis=1)
        box_rois = np.hstack([y1, x1, y2, x2])
        rois[rois_per_box * i:rois_per_box * (i + 1)] = box_rois

    # Generate random ROIs anywhere in the image (10% of count)
    remaining_count = count - (rois_per_box * gt_boxes.shape[0])
    # To avoid generating boxes with zero area, we generate double what
    # we need and filter out the extra. If we get fewer valid boxes
    # than we need, we loop and try again.
    while True:
        y1y2 = np.random.randint(0, image_shape[0], (remaining_count * 2, 2))
        x1x2 = np.random.randint(0, image_shape[1], (remaining_count * 2, 2))
        # Filter out zero area boxes
        threshold = 1
        y1y2 = y1y2[np.abs(y1y2[:, 0] - y1y2[:, 1]) >=
                    threshold][:remaining_count]
        x1x2 = x1x2[np.abs(x1x2[:, 0] - x1x2[:, 1]) >=
                    threshold][:remaining_count]
        if y1y2.shape[0] == remaining_count and x1x2.shape[0] == remaining_count:
            break

    # Sort on axis 1 to ensure x1 <= x2 and y1 <= y2 and then reshape
    # into x1, y1, x2, y2 order
    x1, x2 = np.split(np.sort(x1x2, axis=1), 2, axis=1)
```

```
    y1, y2 = np.split(np.sort(y1y2, axis=1), 2, axis=1)
    global_rois = np.hstack([y1, x1, y2, x2])
    rois[-remaining_count:] = global_rois
    return rois


def data_generator(dataset, config, shuffle=True, augment=False, augmentation=None,
                   random_rois=0, batch_size=1, detection_targets=False,
                   no_augmentation_sources=None):
    """A generator that returns images and corresponding target class ids,
    bounding box deltas, and masks.

    dataset: The Dataset object to pick data from
    config: The model config object
    shuffle: If True, shuffles the samples before every epoch
    augment: (deprecated. Use augmentation instead). If true, apply random
        image augmentation. Currently, only horizontal flipping is offered.
    augmentation: Optional. An imgaug (https://github.com/aleju/imgaug) augmentation.
        For example, passing imgaug.augmenters.Fliplr(0.5) flips images
        right/left 50% of the time.
    random_rois: If > 0 then generate proposals to be used to train the
                 network classifier and mask heads. Useful if training
                 the Mask RCNN part without the RPN.
    batch_size: How many images to return in each call
    detection_targets: If True, generate detection targets (class IDs, bbox
        deltas, and masks). Typically for debugging or visualizations because
        in trainig detection targets are generated by DetectionTargetLayer.
    no_augmentation_sources: Optional. List of sources to exclude for
        augmentation. A source is string that identifies a dataset and is
        defined in the Dataset class.

    Returns a Python generator. Upon calling next() on it, the
    generator returns two lists, inputs and outputs. The contents
    of the lists differs depending on the received arguments:
    inputs list:
    - images: [batch, H, W, C]
    - image_meta: [batch, (meta data)] Image details. See compose_image_meta()
    - rpn_match: [batch, N] Integer (1=positive anchor, -1=negative, 0=neutral)
    - rpn_bbox: [batch, N, (dy, dx, log(dh), log(dw))] Anchor bbox deltas.
    - gt_class_ids: [batch, MAX_GT_INSTANCES] Integer class IDs
    - gt_boxes: [batch, MAX_GT_INSTANCES, (y1, x1, y2, x2)]
    - gt_masks: [batch, height, width, MAX_GT_INSTANCES]. The height and width
                are those of the image unless use_mini_mask is True, in which
                case they are defined in MINI_MASK_SHAPE.

    outputs list: Usually empty in regular training. But if detection_targets
        is True then the outputs list contains target class_ids, bbox deltas,
        and masks.
```

```
    """
    b = 0  # batch item index
    image_index = -1
    image_ids = np.copy(dataset.image_ids)
    error_count = 0
    no_augmentation_sources = no_augmentation_sources or []

    # Anchors
    # [anchor_count, (y1, x1, y2, x2)]
    backbone_shapes = compute_backbone_shapes(config, config.IMAGE_SHAPE)
    anchors = utils.generate_pyramid_anchors(config.RPN_ANCHOR_SCALES,
                                             config.RPN_ANCHOR_RATIOS,
                                             backbone_shapes,
                                             config.BACKBONE_STRIDES,
                                             config.RPN_ANCHOR_STRIDE)

    # Keras requires a generator to run indefinitely.
    while True:
        try:
            # Increment index to pick next image. Shuffle if at the start of an epoch.
            image_index = (image_index + 1) % len(image_ids)
            if shuffle and image_index == 0:
                np.random.shuffle(image_ids)

            # Get GT bounding boxes and masks for image.
            image_id = image_ids[image_index]

            # If the image source is not to be augmented pass None as augmentation
            if dataset.image_info[image_id]['source'] in no_augmentation_sources:
                image, image_meta, gt_class_ids, gt_boxes, gt_masks = \
                load_image_gt(dataset, config, image_id, augment=augment,
                              augmentation=None,
                              use_mini_mask=config.USE_MINI_MASK)
            else:
                image, image_meta, gt_class_ids, gt_boxes, gt_masks = \
                    load_image_gt(dataset, config, image_id, augment=augment,
                                  augmentation=augmentation,
                                  use_mini_mask=config.USE_MINI_MASK)

            # Skip images that have no instances. This can happen in cases
            # where we train on a subset of classes and the image doesn't
            # have any of the classes we care about.
            if not np.any(gt_class_ids > 0):
                continue

            # RPN Targets
            rpn_match, rpn_bbox = build_rpn_targets(image.shape, anchors,
                                                    gt_class_ids, gt_boxes, config)
```

```
# Mask R-CNN Targets
if random_rois:
    rpn_rois = generate_random_rois(
        image.shape, random_rois, gt_class_ids, gt_boxes)
    if detection_targets:
        rois, mrcnn_class_ids, mrcnn_bbox, mrcnn_mask =\
            build_detection_targets(
                rpn_rois, gt_class_ids, gt_boxes, gt_masks, config)

# Init batch arrays
if b == 0:
    batch_image_meta = np.zeros(
        (batch_size,) + image_meta.shape, dtype=image_meta.dtype)
    batch_rpn_match = np.zeros(
        [batch_size, anchors.shape[0], 1], dtype=rpn_match.dtype)
    batch_rpn_bbox = np.zeros(
        [batch_size, config.RPN_TRAIN_ANCHORS_PER_IMAGE, 4], dtype=rpn_bbox.dtype)
    batch_images = np.zeros(
        (batch_size,) + image.shape, dtype=np.float32)
    batch_gt_class_ids = np.zeros(
        (batch_size, config.MAX_GT_INSTANCES), dtype=np.int32)
    batch_gt_boxes = np.zeros(
        (batch_size, config.MAX_GT_INSTANCES, 4), dtype=np.int32)
    batch_gt_masks = np.zeros(
        (batch_size, gt_masks.shape[0], gt_masks.shape[1],
         config.MAX_GT_INSTANCES), dtype=gt_masks.dtype)
    if random_rois:
        batch_rpn_rois = np.zeros(
            (batch_size, rpn_rois.shape[0], 4), dtype=rpn_rois.dtype)
        if detection_targets:
            batch_rois = np.zeros(
                (batch_size,) + rois.shape, dtype=rois.dtype)
            batch_mrcnn_class_ids = np.zeros(
                (batch_size,) + mrcnn_class_ids.shape, dtype=mrcnn_class_ids.dtype)
            batch_mrcnn_bbox = np.zeros(
                (batch_size,) + mrcnn_bbox.shape, dtype=mrcnn_bbox.dtype)
            batch_mrcnn_mask = np.zeros(
                (batch_size,) + mrcnn_mask.shape, dtype=mrcnn_mask.dtype)

# If more instances than fits in the array, sub-sample from them.
if gt_boxes.shape[0] > config.MAX_GT_INSTANCES:
    ids = np.random.choice(
        np.arange(gt_boxes.shape[0]), config.MAX_GT_INSTANCES, replace=False)
    gt_class_ids = gt_class_ids[ids]
    gt_boxes = gt_boxes[ids]
    gt_masks = gt_masks[:, :, ids]
```

```
            # Add to batch
            batch_image_meta[b] = image_meta
            batch_rpn_match[b] = rpn_match[:, np.newaxis]
            batch_rpn_bbox[b] = rpn_bbox
            batch_images[b] = mold_image(image.astype(np.float32), config)
            batch_gt_class_ids[b, :gt_class_ids.shape[0]] = gt_class_ids
            batch_gt_boxes[b, :gt_boxes.shape[0]] = gt_boxes
            batch_gt_masks[b, :, :, :gt_masks.shape[-1]] = gt_masks
            if random_rois:
                batch_rpn_rois[b] = rpn_rois
                if detection_targets:
                    batch_rois[b] = rois
                    batch_mrcnn_class_ids[b] = mrcnn_class_ids
                    batch_mrcnn_bbox[b] = mrcnn_bbox
                    batch_mrcnn_mask[b] = mrcnn_mask
            b += 1

            # Batch full?
            if b >= batch_size:
                inputs = [batch_images, batch_image_meta, batch_rpn_match, batch_rpn_bbox,
                          batch_gt_class_ids, batch_gt_boxes, batch_gt_masks]
                outputs = []

                if random_rois:
                    inputs.extend([batch_rpn_rois])
                    if detection_targets:
                        inputs.extend([batch_rois])
                        # Keras requires that output and targets have the same number of dimensions
                        batch_mrcnn_class_ids = np.expand_dims(
                            batch_mrcnn_class_ids, -1)
                        outputs.extend(
                            [batch_mrcnn_class_ids, batch_mrcnn_bbox, batch_mrcnn_mask])

                yield inputs, outputs

                # start a new batch
                b = 0
        except (GeneratorExit, KeyboardInterrupt):
            raise
        except:
            # Log it and skip the image
            logging.exception("Error processing image {}".format(
                dataset.image_info[image_id]))
            error_count += 1
            if error_count > 5:
                raise
```

```
############################################################
#  MaskRCNN Class
############################################################

class MaskRCNN():
    """Encapsulates the Mask RCNN model functionality.

    The actual Keras model is in the keras_model property.
    """

    def __init__(self, mode, config, model_dir):
        """
        mode: Either "training" or "inference"
        config: A Sub-class of the Config class
        model_dir: Directory to save training logs and trained weights
        """
        assert mode in ['training', 'inference']
        self.mode = mode
        self.config = config
        self.model_dir = model_dir
        self.set_log_dir()
        self.keras_model = self.build(mode=mode, config=config)

    def build(self, mode, config):
        """Build Mask R-CNN architecture.
            input_shape: The shape of the input image.
            mode: Either "training" or "inference". The inputs and
                outputs of the model differ accordingly.
        """
        assert mode in ['training', 'inference']

        # Image size must be dividable by 2 multiple times
        h, w = config.IMAGE_SHAPE[:2]
        if h / 2**6 != int(h / 2**6) or w / 2**6 != int(w / 2**6):
            raise Exception("Image size must be dividable by 2 at least 6 times "
                            "to avoid fractions when downscaling and upscaling."
                            "For example, use 256, 320, 384, 448, 512, ... etc. ")

        # Inputs
        input_image = KL.Input(
            shape=[None, None, config.IMAGE_SHAPE[2]], name="input_image")
        input_image_meta = KL.Input(shape=[config.IMAGE_META_SIZE],
                                    name="input_image_meta")
        if mode == "training":
            # RPN GT
            input_rpn_match = KL.Input(
                shape=[None, 1], name="input_rpn_match", dtype=tf.int32)
            input_rpn_bbox = KL.Input(
```

```
            shape=[None, 4], name="input_rpn_bbox", dtype=tf.float32)

        # Detection GT (class IDs, bounding boxes, and masks)
        # 1. GT Class IDs (zero padded)
        input_gt_class_ids = KL.Input(
            shape=[None], name="input_gt_class_ids", dtype=tf.int32)
        # 2. GT Boxes in pixels (zero padded)
        # [batch, MAX_GT_INSTANCES, (y1, x1, y2, x2)] in image coordinates
        input_gt_boxes = KL.Input(
            shape=[None, 4], name="input_gt_boxes", dtype=tf.float32)
        # Normalize coordinates
        gt_boxes = KL.Lambda(lambda x: norm_boxes_graph(
            x, K.shape(input_image)[1:3]))(input_gt_boxes)
        # 3. GT Masks (zero padded)
        # [batch, height, width, MAX_GT_INSTANCES]
        if config.USE_MINI_MASK:
            input_gt_masks = KL.Input(
                shape=[config.MINI_MASK_SHAPE[0],
                       config.MINI_MASK_SHAPE[1], None],
                name="input_gt_masks", dtype=bool)
        else:
            input_gt_masks = KL.Input(
                shape=[config.IMAGE_SHAPE[0], config.IMAGE_SHAPE[1], None],
                name="input_gt_masks", dtype=bool)
    elif mode == "inference":
        # Anchors in normalized coordinates
        input_anchors = KL.Input(shape=[None, 4], name="input_anchors")

    # Build the shared convolutional layers.
    # Bottom-up Layers
    # Returns a list of the last layers of each stage, 5 in total.
    # Don't create the thead (stage 5), so we pick the 4th item in the list.
    if callable(config.BACKBONE):
        _, C2, C3, C4, C5 = config.BACKBONE(input_image, stage5=True,
                                            train_bn=config.TRAIN_BN)
    else:
        _, C2, C3, C4, C5 = resnet_graph(input_image, config.BACKBONE,
                                         stage5=True, train_bn=config.TRAIN_BN)
    # Top-down Layers
    # TODO: add assert to varify feature map sizes match what's in config
    P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c5p5')(C5)
    P4 = KL.Add(name="fpn_p4add")([
        KL.UpSampling2D(size=(2, 2), name="fpn_p5upsampled")(P5),
        KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c4p4')(C4)])
    P3 = KL.Add(name="fpn_p3add")([
        KL.UpSampling2D(size=(2, 2), name="fpn_p4upsampled")(P4),
        KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c3p3')(C3)])
    P2 = KL.Add(name="fpn_p2add")([
```

```
    KL.UpSampling2D(size=(2, 2), name="fpn_p3upsampled")(P3),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c2p2')(C2)])
# Attach 3x3 conv to all P layers to get the final feature maps.
P2 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p2")(P2)
P3 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p3")(P3)
P4 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p4")(P4)
P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME", name="fpn_p5")(P5)
# P6 is used for the 5th anchor scale in RPN. Generated by
# subsampling from P5 with stride of 2.
P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)

# Note that P6 is used in RPN, but not in the classifier heads.
rpn_feature_maps = [P2, P3, P4, P5, P6]
mrcnn_feature_maps = [P2, P3, P4, P5]

# Anchors
if mode == "training":
    anchors = self.get_anchors(config.IMAGE_SHAPE)
    # Duplicate across the batch dimension because Keras requires it
    # TODO: can this be optimized to avoid duplicating the anchors?
    anchors = np.broadcast_to(anchors, (config.BATCH_SIZE,) + anchors.shape)
    # A hack to get around Keras's bad support for constants
    anchors = KL.Lambda(lambda x: tf.Variable(anchors), name="anchors")(input_image)
else:
    anchors = input_anchors

# RPN Model
rpn = build_rpn_model(config.RPN_ANCHOR_STRIDE,
                      len(config.RPN_ANCHOR_RATIOS), config.TOP_DOWN_PYRAMID_SIZE)
# Loop through pyramid layers
layer_outputs = []  # list of lists
for p in rpn_feature_maps:
    layer_outputs.append(rpn([p]))
# Concatenate layer outputs
# Convert from list of lists of level outputs to list of lists
# of outputs across levels.
# e.g. [[a1, b1, c1], [a2, b2, c2]] => [[a1, a2], [b1, b2], [c1, c2]]
output_names = ["rpn_class_logits", "rpn_class", "rpn_bbox"]
outputs = list(zip(*layer_outputs))
outputs = [KL.Concatenate(axis=1, name=n)(list(o))
           for o, n in zip(outputs, output_names)]

rpn_class_logits, rpn_class, rpn_bbox = outputs

# Generate proposals
# Proposals are [batch, N, (y1, x1, y2, x2)] in normalized coordinates
# and zero padded.
proposal_count = config.POST_NMS_ROIS_TRAINING if mode == "training"\
```

```
            else config.POST_NMS_ROIS_INFERENCE
        rpn_rois = ProposalLayer(
            proposal_count=proposal_count,
            nms_threshold=config.RPN_NMS_THRESHOLD,
            name="ROI",
            config=config)([rpn_class, rpn_bbox, anchors])

        if mode == "training":
            # Class ID mask to mark class IDs supported by the dataset the image
            # came from.
            active_class_ids = KL.Lambda(
                lambda x: parse_image_meta_graph(x)["active_class_ids"]
                )(input_image_meta)

            if not config.USE_RPN_ROIS:
                # Ignore predicted ROIs and use ROIs provided as an input.
                input_rois = KL.Input(shape=[config.POST_NMS_ROIS_TRAINING, 4],
                                      name="input_roi", dtype=np.int32)
                # Normalize coordinates
                target_rois = KL.Lambda(lambda x: norm_boxes_graph(
                    x, K.shape(input_image)[1:3]))(input_rois)
            else:
                target_rois = rpn_rois

            # Generate detection targets
            # Subsamples proposals and generates target outputs for training
            # Note that proposal class IDs, gt_boxes, and gt_masks are zero
            # padded. Equally, returned rois and targets are zero padded.
            rois, target_class_ids, target_bbox, target_mask =\
                DetectionTargetLayer(config, name="proposal_targets")([
                    target_rois, input_gt_class_ids, gt_boxes, input_gt_masks])

            # Network Heads
            # TODO: verify that this handles zero padded ROIs
            mrcnn_class_logits, mrcnn_class, mrcnn_bbox =\
                fpn_classifier_graph(rois, mrcnn_feature_maps, input_image_meta,
                                     config.POOL_SIZE, config.NUM_CLASSES,
                                     train_bn=config.TRAIN_BN,
                                     fc_layers_size=config.FPN_CLASSIF_FC_LAYERS_SIZE)

            mrcnn_mask = build_fpn_mask_graph(rois, mrcnn_feature_maps,
                                              input_image_meta,
                                              config.MASK_POOL_SIZE,
                                              config.NUM_CLASSES,
                                              train_bn=config.TRAIN_BN)

            # TODO: clean up (use tf.identify if necessary)
            output_rois = KL.Lambda(lambda x: x * 1, name="output_rois")(rois)
```

```
        # Losses
        rpn_class_loss = KL.Lambda(lambda x: rpn_class_loss_graph(*x), name="rpn_class_loss")(
            [input_rpn_match, rpn_class_logits])
        rpn_bbox_loss = KL.Lambda(lambda x: rpn_bbox_loss_graph(config, *x), name="rpn_bbox_loss")(
            [input_rpn_bbox, input_rpn_match, rpn_bbox])
        class_loss = KL.Lambda(lambda x: mrcnn_class_loss_graph(*x), name="mrcnn_class_loss")(
            [target_class_ids, mrcnn_class_logits, active_class_ids])
        bbox_loss = KL.Lambda(lambda x: mrcnn_bbox_loss_graph(*x), name="mrcnn_bbox_loss")(
            [target_bbox, target_class_ids, mrcnn_bbox])
        mask_loss = KL.Lambda(lambda x: mrcnn_mask_loss_graph(*x), name="mrcnn_mask_loss")(
            [target_mask, target_class_ids, mrcnn_mask])

        # Model
        inputs = [input_image, input_image_meta,
                  input_rpn_match, input_rpn_bbox, input_gt_class_ids, input_gt_boxes, input_gt_mas
        if not config.USE_RPN_ROIS:
            inputs.append(input_rois)
        outputs = [rpn_class_logits, rpn_class, rpn_bbox,
                   mrcnn_class_logits, mrcnn_class, mrcnn_bbox, mrcnn_mask,
                   rpn_rois, output_rois,
                   rpn_class_loss, rpn_bbox_loss, class_loss, bbox_loss, mask_loss]
        model = KM.Model(inputs, outputs, name='mask_rcnn')
    else:
        # Network Heads
        # Proposal classifier and BBox regressor heads
        mrcnn_class_logits, mrcnn_class, mrcnn_bbox =\
            fpn_classifier_graph(rpn_rois, mrcnn_feature_maps, input_image_meta,
                                 config.POOL_SIZE, config.NUM_CLASSES,
                                 train_bn=config.TRAIN_BN,
                                 fc_layers_size=config.FPN_CLASSIF_FC_LAYERS_SIZE)

        # Detections
        # output is [batch, num_detections, (y1, x1, y2, x2, class_id, score)] in
        # normalized coordinates
        detections = DetectionLayer(config, name="mrcnn_detection")(
            [rpn_rois, mrcnn_class, mrcnn_bbox, input_image_meta])

        # Create masks for detections
        detection_boxes = KL.Lambda(lambda x: x[..., :4])(detections)
        mrcnn_mask = build_fpn_mask_graph(detection_boxes, mrcnn_feature_maps,
                                          input_image_meta,
                                          config.MASK_POOL_SIZE,
                                          config.NUM_CLASSES,
                                          train_bn=config.TRAIN_BN)

        model = KM.Model([input_image, input_image_meta, input_anchors],
                         [detections, mrcnn_class, mrcnn_bbox,
```

```
                                mrcnn_mask, rpn_rois, rpn_class, rpn_bbox],
                            name='mask_rcnn')

        # Add multi-GPU support.
        if config.GPU_COUNT > 1:
            from mrcnn.parallel_model import ParallelModel
            model = ParallelModel(model, config.GPU_COUNT)

        return model

    def find_last(self):
        """Finds the last checkpoint file of the last trained model in the
        model directory.
        Returns:
            The path of the last checkpoint file
        """
        # Get directory names. Each directory corresponds to a model
        dir_names = next(os.walk(self.model_dir))[1]
        key = self.config.NAME.lower()
        dir_names = filter(lambda f: f.startswith(key), dir_names)
        dir_names = sorted(dir_names)
        if not dir_names:
            import errno
            raise FileNotFoundError(
                errno.ENOENT,
                "Could not find model directory under {}".format(self.model_dir))
        # Pick last directory
        dir_name = os.path.join(self.model_dir, dir_names[-1])
        # Find the last checkpoint
        checkpoints = next(os.walk(dir_name))[2]
        checkpoints = filter(lambda f: f.startswith("mask_rcnn"), checkpoints)
        checkpoints = sorted(checkpoints)
        if not checkpoints:
            import errno
            raise FileNotFoundError(
                errno.ENOENT, "Could not find weight files in {}".format(dir_name))
        checkpoint = os.path.join(dir_name, checkpoints[-1])
        return checkpoint

    def load_weights(self, filepath, by_name=False, exclude=None):
        """Modified version of the corresponding Keras function with
        the addition of multi-GPU support and the ability to exclude
        some layers from loading.
        exclude: list of layer names to exclude
        """
        import h5py
        # Conditional import to support versions of Keras before 2.2
        # TODO: remove in about 6 months (end of 2018)
```

```
        try:
            from keras.engine import saving
        except ImportError:
            # Keras before 2.2 used the 'topology' namespace.
            from keras.engine import topology as saving

        if exclude:
            by_name = True

        if h5py is None:
            raise ImportError('`load_weights` requires h5py.')
        f = h5py.File(filepath, mode='r')
        if 'layer_names' not in f.attrs and 'model_weights' in f:
            f = f['model_weights']

        # In multi-GPU training, we wrap the model. Get layers
        # of the inner model because they have the weights.
        keras_model = self.keras_model
        layers = keras_model.inner_model.layers if hasattr(keras_model, "inner_model")\
            else keras_model.layers

        # Exclude some layers
        if exclude:
            layers = filter(lambda l: l.name not in exclude, layers)

        if by_name:
            saving.load_weights_from_hdf5_group_by_name(f, layers)
        else:
            saving.load_weights_from_hdf5_group(f, layers)
        if hasattr(f, 'close'):
            f.close()

        # Update the log directory
        self.set_log_dir(filepath)

    def get_imagenet_weights(self):
        """Downloads ImageNet trained weights from Keras.
        Returns path to weights file.
        """
        from keras.utils.data_utils import get_file
        TF_WEIGHTS_PATH_NO_TOP = 'https://github.com/fchollet/deep-learning-models/'\
                                 'releases/download/v0.2/'\
                                 'resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5'
        weights_path = get_file('resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5',
                                TF_WEIGHTS_PATH_NO_TOP,
                                cache_subdir='models',
                                md5_hash='a268eb855778b3df3c7506639542a6af')
        return weights_path
```

```
def compile(self, learning_rate, momentum):
    """Gets the model ready for training. Adds losses, regularization, and
    metrics. Then calls the Keras compile() function.
    """
    # Optimizer object
    optimizer = keras.optimizers.SGD(
        lr=learning_rate, momentum=momentum,
        clipnorm=self.config.GRADIENT_CLIP_NORM)
    # Add Losses
    # First, clear previously set losses to avoid duplication
    self.keras_model._losses = []
    self.keras_model._per_input_losses = {}
    loss_names = [
        "rpn_class_loss",  "rpn_bbox_loss",
        "mrcnn_class_loss", "mrcnn_bbox_loss", "mrcnn_mask_loss"]
    for name in loss_names:
        layer = self.keras_model.get_layer(name)
        if layer.output in self.keras_model.losses:
            continue
        loss = (
            tf.reduce_mean(layer.output, keepdims=True)
            * self.config.LOSS_WEIGHTS.get(name, 1.))
        self.keras_model.add_loss(loss)

    # Add L2 Regularization
    # Skip gamma and beta weights of batch normalization layers.
    reg_losses = [
        keras.regularizers.l2(self.config.WEIGHT_DECAY)(w) / tf.cast(tf.size(w), tf.float32)
        for w in self.keras_model.trainable_weights
        if 'gamma' not in w.name and 'beta' not in w.name]
    self.keras_model.add_loss(tf.add_n(reg_losses))

    # Compile
    self.keras_model.compile(
        optimizer=optimizer,
        loss=[None] * len(self.keras_model.outputs))

    # Add metrics for losses
    for name in loss_names:
        if name in self.keras_model.metrics_names:
            continue
        layer = self.keras_model.get_layer(name)
        self.keras_model.metrics_names.append(name)
        loss = (
            tf.reduce_mean(layer.output, keepdims=True)
            * self.config.LOSS_WEIGHTS.get(name, 1.))
        self.keras_model.metrics_tensors.append(loss)
```

```
def set_trainable(self, layer_regex, keras_model=None, indent=0, verbose=1):
    """Sets model layers as trainable if their names match
    the given regular expression.
    """
    # Print message on the first call (but not on recursive calls)
    if verbose > 0 and keras_model is None:
        log("Selecting layers to train")

    keras_model = keras_model or self.keras_model

    # In multi-GPU training, we wrap the model. Get layers
    # of the inner model because they have the weights.
    layers = keras_model.inner_model.layers if hasattr(keras_model, "inner_model")\
        else keras_model.layers

    for layer in layers:
        # Is the layer a model?
        if layer.__class__.__name__ == 'Model':
            print("In model: ", layer.name)
            self.set_trainable(
                layer_regex, keras_model=layer, indent=indent + 4)
            continue

        if not layer.weights:
            continue
        # Is it trainable?
        trainable = bool(re.fullmatch(layer_regex, layer.name))
        # Update layer. If layer is a container, update inner layer.
        if layer.__class__.__name__ == 'TimeDistributed':
            layer.layer.trainable = trainable
        else:
            layer.trainable = trainable
        # Print trainable layer names
        if trainable and verbose > 0:
            log("{}{:20}   ({})".format(" " * indent, layer.name,
                                        layer.__class__.__name__))

def set_log_dir(self, model_path=None):
    """Sets the model log directory and epoch counter.

    model_path: If None, or a format different from what this code uses
        then set a new log directory and start epochs from 0. Otherwise,
        extract the log directory and the epoch counter from the file
        name.
    """
    # Set date and epoch counter as if starting a new model
    self.epoch = 0
```

```
        now = datetime.datetime.now()

        # If we have a model path with date and epochs use them
        if model_path:
            # Continue from we left of. Get epoch and date from the file name
            # A sample model path might look like:
            # \path\to\logs\coco20171029T2315\mask_rcnn_coco_0001.h5 (Windows)
            # /path/to/logs/coco20171029T2315/mask_rcnn_coco_0001.h5 (Linux)
            regex = r".*[/\\][\w-]+(\d{4})(\d{2})(\d{2})T(\d{2})(\d{2})[/\\]mask\_rcnn\_[\w-]+(\d{4})\.
            m = re.match(regex, model_path)
            if m:
                now = datetime.datetime(int(m.group(1)), int(m.group(2)), int(m.group(3)),
                                        int(m.group(4)), int(m.group(5)))
                # Epoch number in file is 1-based, and in Keras code it's 0-based.
                # So, adjust for that then increment by one to start from the next epoch
                self.epoch = int(m.group(6)) - 1 + 1
                print('Re-starting from epoch %d' % self.epoch)

        # Directory for training logs
        self.log_dir = os.path.join(self.model_dir, "{}{:%Y%m%dT%H%M}".format(
            self.config.NAME.lower(), now))

        # Path to save after each epoch. Include placeholders that get filled by Keras.
        self.checkpoint_path = os.path.join(self.log_dir, "mask_rcnn_{}_*epoch*.h5".format(
            self.config.NAME.lower()))
        self.checkpoint_path = self.checkpoint_path.replace(
            "*epoch*", "{epoch:04d}")

    def train(self, train_dataset, val_dataset, learning_rate, epochs, layers,
              augmentation=None, custom_callbacks=None, no_augmentation_sources=None):
        """Train the model.
        train_dataset, val_dataset: Training and validation Dataset objects.
        learning_rate: The learning rate to train with
        epochs: Number of training epochs. Note that previous training epochs
                are considered to be done alreay, so this actually determines
                the epochs to train in total rather than in this particaular
                call.
        layers: Allows selecting wich layers to train. It can be:
            - A regular expression to match layer names to train
            - One of these predefined values:
              heads: The RPN, classifier and mask heads of the network
              all: All the layers
              3+: Train Resnet stage 3 and up
              4+: Train Resnet stage 4 and up
              5+: Train Resnet stage 5 and up
        augmentation: Optional. An imgaug (https://github.com/aleju/imgaug)
            augmentation. For example, passing imgaug.augmenters.Fliplr(0.5)
            flips images right/left 50% of the time. You can pass complex
```

```
        augmentations as well. This augmentation applies 50% of the
        time, and when it does it flips images right/left half the time
        and adds a Gaussian blur with a random sigma in range 0 to 5.

            augmentation = imgaug.augmenters.Sometimes(0.5, [
                imgaug.augmenters.Fliplr(0.5),
                imgaug.augmenters.GaussianBlur(sigma=(0.0, 5.0))
            ])
        custom_callbacks: Optional. Add custom callbacks to be called
            with the keras fit_generator method. Must be list of type keras.callbacks.
    no_augmentation_sources: Optional. List of sources to exclude for
        augmentation. A source is string that identifies a dataset and is
        defined in the Dataset class.
    """
    assert self.mode == "training", "Create model in training mode."

    # Pre-defined layer regular expressions
    layer_regex = {
        # all layers but the backbone
        "heads": r"(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
        # From a specific Resnet stage and up
        "3+": r"(res3.*)|(bn3.*)|(res4.*)|(bn4.*)|(res5.*)|(bn5.*)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)"
        "4+": r"(res4.*)|(bn4.*)|(res5.*)|(bn5.*)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
        "5+": r"(res5.*)|(bn5.*)|(mrcnn\_.*)|(rpn\_.*)|(fpn\_.*)",
        # All layers
        "all": ".*",
    }
    if layers in layer_regex.keys():
        layers = layer_regex[layers]

    # Data generators
    train_generator = data_generator(train_dataset, self.config, shuffle=True,
                                     augmentation=augmentation,
                                     batch_size=self.config.BATCH_SIZE,
                                     no_augmentation_sources=no_augmentation_sources)
    val_generator = data_generator(val_dataset, self.config, shuffle=True,
                                   batch_size=self.config.BATCH_SIZE)

    # Create log_dir if it does not exist
    if not os.path.exists(self.log_dir):
        os.makedirs(self.log_dir)

    # Callbacks
    callbacks = [
        keras.callbacks.TensorBoard(log_dir=self.log_dir,
                                    histogram_freq=0, write_graph=True, write_images=False),
        keras.callbacks.ModelCheckpoint(self.checkpoint_path,
                                        verbose=0, save_weights_only=True),
```

```
        ]

        # Add custom callbacks to the list
        if custom_callbacks:
            callbacks += custom_callbacks

        # Train
        log("\nStarting at epoch {}. LR={}\n".format(self.epoch, learning_rate))
        log("Checkpoint Path: {}".format(self.checkpoint_path))
        self.set_trainable(layers)
        self.compile(learning_rate, self.config.LEARNING_MOMENTUM)

        # Work-around for Windows: Keras fails on Windows when using
        # multiprocessing workers. See discussion here:
        # https://github.com/matterport/Mask_RCNN/issues/13#issuecomment-353124009
        if os.name is 'nt':
            workers = 0
        else:
            workers = multiprocessing.cpu_count()

        self.keras_model.fit_generator(
            train_generator,
            initial_epoch=self.epoch,
            epochs=epochs,
            steps_per_epoch=self.config.STEPS_PER_EPOCH,
            callbacks=callbacks,
            validation_data=val_generator,
            validation_steps=self.config.VALIDATION_STEPS,
            max_queue_size=100,
            workers=workers,
            use_multiprocessing=True,
        )
        self.epoch = max(self.epoch, epochs)

    def mold_inputs(self, images):
        """Takes a list of images and modifies them to the format expected
        as an input to the neural network.
        images: List of image matrices [height,width,depth]. Images can have
            different sizes.

        Returns 3 Numpy matrices:
        molded_images: [N, h, w, 3]. Images resized and normalized.
        image_metas: [N, length of meta data]. Details about each image.
        windows: [N, (y1, x1, y2, x2)]. The portion of the image that has the
            original image (padding excluded).
        """
        molded_images = []
        image_metas = []
```

```
        windows = []
        for image in images:
            # Resize image
            # TODO: move resizing to mold_image()
            molded_image, window, scale, padding, crop = utils.resize_image(
                image,
                min_dim=self.config.IMAGE_MIN_DIM,
                min_scale=self.config.IMAGE_MIN_SCALE,
                max_dim=self.config.IMAGE_MAX_DIM,
                mode=self.config.IMAGE_RESIZE_MODE)
            molded_image = mold_image(molded_image, self.config)
            # Build image_meta
            image_meta = compose_image_meta(
                0, image.shape, molded_image.shape, window, scale,
                np.zeros([self.config.NUM_CLASSES], dtype=np.int32))
            # Append
            molded_images.append(molded_image)
            windows.append(window)
            image_metas.append(image_meta)
        # Pack into arrays
        molded_images = np.stack(molded_images)
        image_metas = np.stack(image_metas)
        windows = np.stack(windows)
        return molded_images, image_metas, windows

    def unmold_detections(self, detections, mrcnn_mask, original_image_shape,
                          image_shape, window):
        """Reformats the detections of one image from the format of the neural
        network output to a format suitable for use in the rest of the
        application.

        detections: [N, (y1, x1, y2, x2, class_id, score)] in normalized coordinates
        mrcnn_mask: [N, height, width, num_classes]
        original_image_shape: [H, W, C] Original image shape before resizing
        image_shape: [H, W, C] Shape of the image after resizing and padding
        window: [y1, x1, y2, x2] Pixel coordinates of box in the image where the real
                image is excluding the padding.

        Returns:
        boxes: [N, (y1, x1, y2, x2)] Bounding boxes in pixels
        class_ids: [N] Integer class IDs for each bounding box
        scores: [N] Float probability scores of the class_id
        masks: [height, width, num_instances] Instance masks
        """
        # How many detections do we have?
        # Detections array is padded with zeros. Find the first class_id == 0.
        zero_ix = np.where(detections[:, 4] == 0)[0]
        N = zero_ix[0] if zero_ix.shape[0] > 0 else detections.shape[0]
```

```
        # Extract boxes, class_ids, scores, and class-specific masks
        boxes = detections[:N, :4]
        class_ids = detections[:N, 4].astype(np.int32)
        scores = detections[:N, 5]
        masks = mrcnn_mask[np.arange(N), :, :, class_ids]

        # Translate normalized coordinates in the resized image to pixel
        # coordinates in the original image before resizing
        window = utils.norm_boxes(window, image_shape[:2])
        wy1, wx1, wy2, wx2 = window
        shift = np.array([wy1, wx1, wy1, wx1])
        wh = wy2 - wy1  # window height
        ww = wx2 - wx1  # window width
        scale = np.array([wh, ww, wh, ww])
        # Convert boxes to normalized coordinates on the window
        boxes = np.divide(boxes - shift, scale)
        # Convert boxes to pixel coordinates on the original image
        boxes = utils.denorm_boxes(boxes, original_image_shape[:2])

        # Filter out detections with zero area. Happens in early training when
        # network weights are still random
        exclude_ix = np.where(
            (boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes[:, 1]) <= 0)[0]
        if exclude_ix.shape[0] > 0:
            boxes = np.delete(boxes, exclude_ix, axis=0)
            class_ids = np.delete(class_ids, exclude_ix, axis=0)
            scores = np.delete(scores, exclude_ix, axis=0)
            masks = np.delete(masks, exclude_ix, axis=0)
            N = class_ids.shape[0]

        # Resize masks to original image size and set boundary threshold.
        full_masks = []
        for i in range(N):
            # Convert neural network mask to full size mask
            full_mask = utils.unmold_mask(masks[i], boxes[i], original_image_shape)
            full_masks.append(full_mask)
        full_masks = np.stack(full_masks, axis=-1)\
            if full_masks else np.empty(original_image_shape[:2] + (0,))

        return boxes, class_ids, scores, full_masks

    def detect(self, images, verbose=0):
        """Runs the detection pipeline.

        images: List of images, potentially of different sizes.

        Returns a list of dicts, one dict per image. The dict contains:
```

```
        rois: [N, (y1, x1, y2, x2)] detection bounding boxes
        class_ids: [N] int class IDs
        scores: [N] float probability scores for the class IDs
        masks: [H, W, N] instance binary masks
        """
        assert self.mode == "inference", "Create model in inference mode."
        assert len(
            images) == self.config.BATCH_SIZE, "len(images) must be equal to BATCH_SIZE"

        if verbose:
            log("Processing {} images".format(len(images)))
            for image in images:
                log("image", image)

        # Mold inputs to format expected by the neural network
        molded_images, image_metas, windows = self.mold_inputs(images)

        # Validate image sizes
        # All images in a batch MUST be of the same size
        image_shape = molded_images[0].shape
        for g in molded_images[1:]:
            assert g.shape == image_shape,\
                "After resizing, all images must have the same size. Check IMAGE_RESIZE_MODE and image

        # Anchors
        anchors = self.get_anchors(image_shape)
        # Duplicate across the batch dimension because Keras requires it
        # TODO: can this be optimized to avoid duplicating the anchors?
        anchors = np.broadcast_to(anchors, (self.config.BATCH_SIZE,) + anchors.shape)

        if verbose:
            log("molded_images", molded_images)
            log("image_metas", image_metas)
            log("anchors", anchors)
        # Run object detection
        detections, _, _, mrcnn_mask, _, _, _ =\
            self.keras_model.predict([molded_images, image_metas, anchors], verbose=0)
        # Process detections
        results = []
        for i, image in enumerate(images):
            final_rois, final_class_ids, final_scores, final_masks =\
                self.unmold_detections(detections[i], mrcnn_mask[i],
                                       image.shape, molded_images[i].shape,
                                       windows[i])
            results.append({
                "rois": final_rois,
                "class_ids": final_class_ids,
                "scores": final_scores,
```

```
                "masks": final_masks,
            })
        return results

    def detect_molded(self, molded_images, image_metas, verbose=0):
        """Runs the detection pipeline, but expect inputs that are
        molded already. Used mostly for debugging and inspecting
        the model.

        molded_images: List of images loaded using load_image_gt()
        image_metas: image meta data, also returned by load_image_gt()

        Returns a list of dicts, one dict per image. The dict contains:
        rois: [N, (y1, x1, y2, x2)] detection bounding boxes
        class_ids: [N] int class IDs
        scores: [N] float probability scores for the class IDs
        masks: [H, W, N] instance binary masks
        """
        assert self.mode == "inference", "Create model in inference mode."
        assert len(molded_images) == self.config.BATCH_SIZE,\
            "Number of images must be equal to BATCH_SIZE"

        if verbose:
            log("Processing {} images".format(len(molded_images)))
            for image in molded_images:
                log("image", image)

        # Validate image sizes
        # All images in a batch MUST be of the same size
        image_shape = molded_images[0].shape
        for g in molded_images[1:]:
            assert g.shape == image_shape, "Images must have the same size"

        # Anchors
        anchors = self.get_anchors(image_shape)
        # Duplicate across the batch dimension because Keras requires it
        # TODO: can this be optimized to avoid duplicating the anchors?
        anchors = np.broadcast_to(anchors, (self.config.BATCH_SIZE,) + anchors.shape)

        if verbose:
            log("molded_images", molded_images)
            log("image_metas", image_metas)
            log("anchors", anchors)
        # Run object detection
        detections, _, _, mrcnn_mask, _, _, _ =\
            self.keras_model.predict([molded_images, image_metas, anchors], verbose=0)
        # Process detections
        results = []
```

```
        for i, image in enumerate(molded_images):
            window = [0, 0, image.shape[0], image.shape[1]]
            final_rois, final_class_ids, final_scores, final_masks =\
                self.unmold_detections(detections[i], mrcnn_mask[i],
                                       image.shape, molded_images[i].shape,
                                       window)
            results.append({
                "rois": final_rois,
                "class_ids": final_class_ids,
                "scores": final_scores,
                "masks": final_masks,
            })
        return results

    def get_anchors(self, image_shape):
        """Returns anchor pyramid for the given image size."""
        backbone_shapes = compute_backbone_shapes(self.config, image_shape)
        # Cache anchors and reuse if image shape is the same
        if not hasattr(self, "_anchor_cache"):
            self._anchor_cache = {}
        if not tuple(image_shape) in self._anchor_cache:
            # Generate Anchors
            a = utils.generate_pyramid_anchors(
                self.config.RPN_ANCHOR_SCALES,
                self.config.RPN_ANCHOR_RATIOS,
                backbone_shapes,
                self.config.BACKBONE_STRIDES,
                self.config.RPN_ANCHOR_STRIDE)
            # Keep a copy of the latest anchors in pixel coordinates because
            # it's used in inspect_model notebooks.
            # TODO: Remove this after the notebook are refactored to not use it
            self.anchors = a
            # Normalize coordinates
            self._anchor_cache[tuple(image_shape)] = utils.norm_boxes(a, image_shape[:2])
        return self._anchor_cache[tuple(image_shape)]

    def ancestor(self, tensor, name, checked=None):
        """Finds the ancestor of a TF tensor in the computation graph.
        tensor: TensorFlow symbolic tensor.
        name: Name of ancestor tensor to find
        checked: For internal use. A list of tensors that were already
                 searched to avoid loops in traversing the graph.
        """
        checked = checked if checked is not None else []
        # Put a limit on how deep we go to avoid very long loops
        if len(checked) > 500:
            return None
        # Convert name to a regex and allow matching a number prefix
```

```
        # because Keras adds them automatically
        if isinstance(name, str):
            name = re.compile(name.replace("/", r"(\_\d+)*/"))

        parents = tensor.op.inputs
        for p in parents:
            if p in checked:
                continue
            if bool(re.fullmatch(name, p.name)):
                return p
            checked.append(p)
            a = self.ancestor(p, name, checked)
            if a is not None:
                return a
        return None

    def find_trainable_layer(self, layer):
        """If a layer is encapsulated by another layer, this function
        digs through the encapsulation and returns the layer that holds
        the weights.
        """
        if layer.__class__.__name__ == 'TimeDistributed':
            return self.find_trainable_layer(layer.layer)
        return layer

    def get_trainable_layers(self):
        """Returns a list of layers that have weights."""
        layers = []
        # Loop through all layers
        for l in self.keras_model.layers:
            # If layer is a wrapper, find inner trainable layer
            l = self.find_trainable_layer(l)
            # Include layer if it has weights
            if l.get_weights():
                layers.append(l)
        return layers

    def run_graph(self, images, outputs, image_metas=None):
        """Runs a sub-set of the computation graph that computes the given
        outputs.

        image_metas: If provided, the images are assumed to be already
            molded (i.e. resized, padded, and normalized)

        outputs: List of tuples (name, tensor) to compute. The tensors are
            symbolic TensorFlow tensors and the names are for easy tracking.

        Returns an ordered dict of results. Keys are the names received in the
```

```
        input and values are Numpy arrays.
        """
        model = self.keras_model

        # Organize desired outputs into an ordered dict
        outputs = OrderedDict(outputs)
        for o in outputs.values():
            assert o is not None

        # Build a Keras function to run parts of the computation graph
        inputs = model.inputs
        if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
            inputs += [K.learning_phase()]
        kf = K.function(model.inputs, list(outputs.values()))

        # Prepare inputs
        if image_metas is None:
            molded_images, image_metas, _ = self.mold_inputs(images)
        else:
            molded_images = images
        image_shape = molded_images[0].shape
        # Anchors
        anchors = self.get_anchors(image_shape)
        # Duplicate across the batch dimension because Keras requires it
        # TODO: can this be optimized to avoid duplicating the anchors?
        anchors = np.broadcast_to(anchors, (self.config.BATCH_SIZE,) + anchors.shape)
        model_in = [molded_images, image_metas, anchors]

        # Run inference
        if model.uses_learning_phase and not isinstance(K.learning_phase(), int):
            model_in.append(0.)
        outputs_np = kf(model_in)

        # Pack the generated Numpy arrays into a a dict and log the results.
        outputs_np = OrderedDict([(k, v)
                                  for k, v in zip(outputs.keys(), outputs_np)])
        for k, v in outputs_np.items():
            log(k, v)
        return outputs_np


############################################################
#  Data Formatting
############################################################

def compose_image_meta(image_id, original_image_shape, image_shape,
                       window, scale, active_class_ids):
    """Takes attributes of an image and puts them in one 1D array.
```

```
    image_id: An int ID of the image. Useful for debugging.
    original_image_shape: [H, W, C] before resizing or padding.
    image_shape: [H, W, C] after resizing and padding
    window: (y1, x1, y2, x2) in pixels. The area of the image where the real
            image is (excluding the padding)
    scale: The scaling factor applied to the original image (float32)
    active_class_ids: List of class_ids available in the dataset from which
        the image came. Useful if training on images from multiple datasets
        where not all classes are present in all datasets.
    """
    meta = np.array(
        [image_id] +                  # size=1
        list(original_image_shape) +  # size=3
        list(image_shape) +           # size=3
        list(window) +                # size=4 (y1, x1, y2, x2) in image cooredinates
        [scale] +                     # size=1
        list(active_class_ids)        # size=num_classes
    )
    return meta


def parse_image_meta(meta):
    """Parses an array that contains image attributes to its components.
    See compose_image_meta() for more details.

    meta: [batch, meta length] where meta length depends on NUM_CLASSES

    Returns a dict of the parsed values.
    """
    image_id = meta[:, 0]
    original_image_shape = meta[:, 1:4]
    image_shape = meta[:, 4:7]
    window = meta[:, 7:11]  # (y1, x1, y2, x2) window of image in in pixels
    scale = meta[:, 11]
    active_class_ids = meta[:, 12:]
    return {
        "image_id": image_id.astype(np.int32),
        "original_image_shape": original_image_shape.astype(np.int32),
        "image_shape": image_shape.astype(np.int32),
        "window": window.astype(np.int32),
        "scale": scale.astype(np.float32),
        "active_class_ids": active_class_ids.astype(np.int32),
    }


def parse_image_meta_graph(meta):
    """Parses a tensor that contains image attributes to its components.
```

```
    See compose_image_meta() for more details.

    meta: [batch, meta length] where meta length depends on NUM_CLASSES

    Returns a dict of the parsed tensors.
    """
    image_id = meta[:, 0]
    original_image_shape = meta[:, 1:4]
    image_shape = meta[:, 4:7]
    window = meta[:, 7:11]  # (y1, x1, y2, x2) window of image in in pixels
    scale = meta[:, 11]
    active_class_ids = meta[:, 12:]
    return {
        "image_id": image_id,
        "original_image_shape": original_image_shape,
        "image_shape": image_shape,
        "window": window,
        "scale": scale,
        "active_class_ids": active_class_ids,
    }


def mold_image(images, config):
    """Expects an RGB image (or array of images) and subtracts
    the mean pixel and converts it to float. Expects image
    colors in RGB order.
    """
    return images.astype(np.float32) - config.MEAN_PIXEL


def unmold_image(normalized_images, config):
    """Takes a image normalized with mold() and returns the original."""
    return (normalized_images + config.MEAN_PIXEL).astype(np.uint8)


############################################################
#  Miscellenous Graph Functions
############################################################

def trim_zeros_graph(boxes, name='trim_zeros'):
    """Often boxes are represented with matrices of shape [N, 4] and
    are padded with zeros. This removes zero boxes.

    boxes: [N, 4] matrix of boxes.
    non_zeros: [N] a 1D boolean mask identifying the rows to keep
    """
    non_zeros = tf.cast(tf.reduce_sum(tf.abs(boxes), axis=1), tf.bool)
    boxes = tf.boolean_mask(boxes, non_zeros, name=name)
```

```
    return boxes, non_zeros


def batch_pack_graph(x, counts, num_rows):
    """Picks different number of values from each row
    in x depending on the values in counts.
    """
    outputs = []
    for i in range(num_rows):
        outputs.append(x[i, :counts[i]])
    return tf.concat(outputs, axis=0)


def norm_boxes_graph(boxes, shape):
    """Converts boxes from pixel coordinates to normalized coordinates.
    boxes: [..., (y1, x1, y2, x2)] in pixel coordinates
    shape: [..., (height, width)] in pixels

    Note: In pixel coordinates (y2, x2) is outside the box. But in normalized
    coordinates it's inside the box.

    Returns:
        [..., (y1, x1, y2, x2)] in normalized coordinates
    """
    h, w = tf.split(tf.cast(shape, tf.float32), 2)
    scale = tf.concat([h, w, h, w], axis=-1) - tf.constant(1.0)
    shift = tf.constant([0., 0., 1., 1.])
    return tf.divide(boxes - shift, scale)


def denorm_boxes_graph(boxes, shape):
    """Converts boxes from normalized coordinates to pixel coordinates.
    boxes: [..., (y1, x1, y2, x2)] in normalized coordinates
    shape: [..., (height, width)] in pixels

    Note: In pixel coordinates (y2, x2) is outside the box. But in normalized
    coordinates it's inside the box.

    Returns:
        [..., (y1, x1, y2, x2)] in pixel coordinates
    """
    h, w = tf.split(tf.cast(shape, tf.float32), 2)
    scale = tf.concat([h, w, h, w], axis=-1) - tf.constant(1.0)
    shift = tf.constant([0., 0., 1., 1.])
    return tf.cast(tf.round(tf.multiply(boxes, scale) + shift), tf.int32)
```

Appendix E

CDRD Model Training and Testing Code

```
# Commented out IPython magic to ensure Python compatibility.
import os
import json
import sys
from mrcnn.config import Config
import skimage.draw
import random
import math
import re
import time
import numpy as np

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.image as mpimg
import tensorflow as tf
# Root directory of the project
ROOT_DIR = os.path.abspath(r"C:\Users\Saad Masrur Rao\Downloads
\wastedata-Mask_RCNN-multiple-classes-master
\wastedata-Mask_RCNN-multiple-classes-master
\main\Mask_RCNN")

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

```
from mrcnn.model import log

# %matplotlib inline

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Path to trained weights
# You can download this file from the Releases page
# https://github.com/matterport/Mask_RCNN/releases
WEIGHTS_PATH = "mask_rcnn_object_0100.h5"  # TODO: update this path
class CustomConfig(Config):
    """Configuration for training on the toy  dataset.
    Derives from the base Config class and overrides some values.
    """
    # Give the configuration a recognizable name
    NAME = "object"

    # We use a GPU with 12GB memory, which can fit two images.
    # Adjust down if you use a smaller GPU.
    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 11  # Background + toy

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 100

    # Skip detections with < 90% confidence
    DETECTION_MIN_CONFIDENCE = 0.9


class CustomDataset(utils.Dataset):

    def load_custom(self, dataset_dir, subset):
        """Load a subset of the Dog-Cat dataset.
        dataset_dir: Root directory of the dataset.
        subset: Subset to load: train or val
        """
        # Add classes. We have only one class to add.
        self.add_class("object", 1, "1")
        self.add_class("object", 2, "2")
        self.add_class("object", 3, "3")
        self.add_class("object", 4, "4")
        self.add_class("object", 5, "5")
        self.add_class("object", 6, "6")
        self.add_class("object", 7, "7")
        self.add_class("object", 8, "8")
```

```
self.add_class("object", 9, "9")
self.add_class("object", 10, "0")
self.add_class("object", 11, "point")
# self.add_class("object", 3, "xyz")

# Train or validation dataset?
assert subset in ["train", "val"]
dataset_dir = os.path.join(dataset_dir, subset)

# Load annotations
# VGG Image Annotator saves each image in the form:
# { 'filename': '28503151_5b5b7ec140_b.jpg',
#   'regions': {
#       '0': {
#           'region_attributes': {},
#           'shape_attributes': {
#               'all_points_x': [...],
#               'all_points_y': [...],
#               'name': 'polygon'}},
#       ... more regions ...
#   },
#   'size': 100202
# }
# We mostly care about the x and y coordinates of each region
annotations1 = json.load(open(os.path.join(dataset_dir, "via_project.json")))
# print(annotations1)
annotations = list(annotations1.values())  # don't need the dict keys

# The VIA tool saves images in the JSON even if they don't have any
# annotations. Skip unannotated images.
annotations = [a for a in annotations if a['regions']]

# Add images
for a in annotations:
    # print(a)
    # Get the x, y coordinaets of points of the polygons that make up
    # the outline of each object instance. There are stores in the
    # shape_attributes (see json format above)
    polygons = [r['shape_attributes'] for r in a['regions']]
    objects = [s['region_attributes']['name'] for s in a['regions']]
    print("objects:", objects)
    name_dict = {'1': "1", '2': "2", '3': "3",
    '4': "4", '5': "5", '6': "6",
    '7': "7", '8': "8", '9': "9",
      '0': "10",'point':"11"}  # ,"xyz": 3}
    # key = tuple(name_dict)
    num_ids = [name_dict[a] for a in objects]
```

```
            # num_ids = [int(n['Event']) for n in objects]
            # load_mask() needs the image size to convert polygons to masks.
            # Unfortunately, VIA doesn't include it in JSON, so we must read
            # the image. This is only managable since the dataset is tiny.
            print("numids", num_ids)
            image_path = os.path.join(dataset_dir, a['filename'])
            image = skimage.io.imread(image_path)
            height, width = image.shape[:2]

            self.add_image(
                "object",  ## for a single class just add the name here
                image_id=a['filename'],  # use file name as a unique image id
                path=image_path,
                width=width, height=height,
                polygons=polygons,
                num_ids=num_ids
            )
config = CustomConfig()
CUSTOM_DIR = os.path.join(ROOT_DIR, "dataset")


class InferenceConfig(config.__class__):
    # Run detection on one image at a time
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    DETECTION_MIN_CONFIDENCE = 0.7


config = InferenceConfig()
config.display()

# Device to load the neural network on.
# Useful if you're training a model on the same
# machine, in which case use CPU and leave the
# GPU for training.
DEVICE = "/gpu:0"  # /cpu:0 or /gpu:0

# Inspect the model in training or inference modes
# values: 'inference' or 'training'
# TODO: code for 'training' test mode not ready yet
TEST_MODE = "inference"


def get_ax(rows=1, cols=1, size=16):
    """Return a Matplotlib Axes array to be used in
    all visualizations in the notebook. Provide a
    central point to control graph sizes.
```

```
    Adjust the size attribute to control how big to render images
    """
    _, ax = plt.subplots(rows, cols, figsize=(size * cols, size * rows))
    return ax


# Load validation dataset
CUSTOM_DIR = "dataset"
dataset = CustomDataset()

# Load validation dataset
CUSTOM_DIR = "dataset"
dataset = CustomDataset()
dataset.load_custom(CUSTOM_DIR, "val")

# Must call before using the dataset
dataset.prepare()

print("Images: {}\nClasses: {}".format(len(dataset.image_ids), dataset.class_names))

# LOAD MODEL
# Create model in inference mode
with tf.device(DEVICE):
    model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR,
                              config=config)

# Load COCO weights, Or load the last model you trained
weights_path = WEIGHTS_PATH

# Load weights
print("Loading weights ", weights_path)
model.load_weights(weights_path, by_name=True)

# RUN DETECTION
image_id = random.choice(dataset.image_ids)
print(image_id)
image, image_meta, gt_class_id, gt_bbox, gt_mask = \
    modellib.load_image_gt(dataset, config, image_id, use_mini_mask=False)
info = dataset.image_info[image_id]
print("image ID: {}.{} ({}) {}".format(info["source"], info["id"], image_id,
                                       dataset.image_reference(image_id)))

# Run object detection
results = model.detect([image], verbose=1)

# Display results
ax = get_ax(1)
r = results[0]
```

```
visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
                            dataset.class_names, r['scores'], ax=ax,
                            title="Predictions")
log("gt_class_id", gt_class_id)
log("gt_bbox", gt_bbox)
log("gt_mask", gt_mask)

# This is for predicting images which are not present in dataset
# image_id = random.choice(dataset.image_ids)
image1 = mpimg.imread('s2354.jpg')

# Run object detection
print(len([image1]))
results1 = model.detect([image1], verbose=1)

# Display results
ax = get_ax(1)
r1 = results1[0]

print(r1['class_ids'])

visualize.display_instances(image1, r1['rois'], r1['masks'], r1['class_ids'],
                            dataset.class_names, r1['scores'], ax=ax,
                            title="Predictions1")
plt.show()
id=r1['class_ids']

N = id.shape[0]
print(N)
conid=[]
for i in range(N):
  conid.append(id[i])
print(conid)
boxes=r1['rois']
N = boxes.shape[0]
print(N)
a=[]
for i in range(N):
 print(boxes[i])
 y1, x1, y2, x2 = boxes[i]
 print(x1)
 a.append(x1)
#for i in range(N):
print(a)
xval=a
print(xval)
names =xval
print(xval)
```

```
professions =conid
print(conid)
professions_dict = {}
for i in range(len(names)):
    professions_dict[names[i]] = professions[i]
print(professions_dict)
a=xval
a.sort()
print(a)
num_ids = [professions_dict[b] for b in a]
print(num_ids)
digits=num_ids
name= {1: 1,2: 2,3:3,4:4,5:5,6:6,7:7,8:8,9:9,10:0,11:'.'} #,"xyz": 3}
numf = [name[b] for b in digits]
translation = {39: None}

wrong = [name[b] for b in id]
translation = {39: None}
print('saad')
print(wrong)

wron = str(wrong).translate(translation)
wron=wron.replace(',', '')
print(wron)


# Printing list using translate Method
dig1 = str(numf).translate(translation)
dig=dig1.replace(',', '')
print(dig)
```

Appendix F

Main Code

```
# Commented out IPython magic to ensure Python compatibility.
from __future__ import division, print_function
# coding=utf-8
import sys
import os
import glob
import re
import numpy as np
import math
import os
import cv2
import json
import sys
from mrcnn.config import Config
import skimage.draw
import random
import math
import re
import time
import numpy as np
from PIL import Image
import PIL
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.image as mpimg
import tensorflow as tf
#code of app


# Keras
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
```

```
from keras.models import load_model
from keras.preprocessing import image

# Flask utils
from flask import Flask, redirect, url_for, request, render_template
from werkzeug.utils import secure_filename
from gevent.pywsgi import WSGIServer
from fpdf import FPDF
import pandas as pd
import random
from pdf_mail import sendpdf
import numpy as np
import matplotlib.pyplot as plt



############################################################
# Root directory of the project
ROOT_DIR = os.path.abspath(r"C:\Users\Saad Masrur Rao\Downloads\wastedata-Mask_RCNN-multiple-classes-ma

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

# %matplotlib inline
df=pd.read_csv('bill.csv' )
id=df["Meter I'D"]
p_id=(random.choice(id))
index=np.where(df["Meter I'D"] == p_id)
index=(index[0])
pmr=np.array(df.iloc[index,1]).flatten()
pmr=(pmr[0])

uem=np.array(df.iloc[index,2])
vem=(uem[0])
print(vem)

peter='Meter id is                                    : '
m_id=peter+str(p_id)
print(m_id)
jazz='Previous Meter Reading is          : '
p_m_r1=jazz+str(pmr)

name=np.array(df.iloc[index,3])
name=(name[0])
```

```
sapro='                    Consumer Name : '
sap=sapro+str(name+'                              '            )

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")
# Define a flask app
app = Flask(__name__)
# Path to trained weights
# You can download this file from the Releases page
# https://github.com/matterport/Mask_RCNN/releases
WEIGHTS_PATH = "mask_rcnn_object_0100.h5"  # TODO: update this path
class CustomConfig(Config):
    """Configuration for training on the toy  dataset.
    Derives from the base Config class and overrides some values.
    """
    # Give the configuration a recognizable name
    NAME = "object"

    # We use a GPU with 12GB memory, which can fit two images.
    # Adjust down if you use a smaller GPU.
    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 11  # Background + toy

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 100

    # Skip detections with < 90% confidence
    DETECTION_MIN_CONFIDENCE = 0.9



config = CustomConfig()
CUSTOM_DIR = os.path.join(ROOT_DIR, "dataset")


class InferenceConfig(config.__class__):
    # Run detection on one image at a time
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    DETECTION_MIN_CONFIDENCE = 0.7


config = InferenceConfig()
config.display()
```

```
DEVICE = "/gpu:0"  # /cpu:0 or /gpu:0




def get_ax(rows=1, cols=1, size=16):
    """Return a Matplotlib Axes array to be used in
    all visualizations in the notebook. Provide a
    central point to control graph sizes.

    Adjust the size attribute to control how big to render images
    """
    _, ax = plt.subplots(rows, cols, figsize=(size * cols, size * rows))
    return ax



# LOAD MODEL
# Create model in inference mode
with tf.device(DEVICE):
    model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR,
                              config=config)

# Load COCO weights, Or load the last model you trained
weights_path = WEIGHTS_PATH

# Load weights
print("Loading weights ", weights_path)
model.load_weights(weights_path, by_name=True)
model.keras_model._make_predict_function()
print('Model loaded. Check https://fyp.skynetjoe.com/')

# This is for predicting images which are not present in dataset
# image_id = random.choice(dataset.image_ids)
#define a funtion
def model_predict(img_path, model):
    path = r'C:\Artificial Intelligence\testmaskrcnn'
    image1 = mpimg.imread(img_path)
    img = cv2.resize(image1, (300,210))
    cv2.imwrite(os.path.join(path, 'meter.jpg'),img )
    # Run object detection
    results1 = model.detect([image1], verbose=1)
    ax = get_ax(1)
    r1 = results1[0]
    id = r1['class_ids']
    N = id.shape[0]
    conid = []
    for i in range(N):
```

```
            conid.append(id[i])
        boxes = r1['rois']
        N = boxes.shape[0]
        a = []
        for i in range(N):
            y1, x1, y2, x2 = boxes[i]
            a.append(x1)
        # for i in range(N):
        xval = a
        names = xval
        professions = conid
        professions_dict = {}
        for i in range(len(names)):
            professions_dict[names[i]] = professions[i]
        a = xval
        a.sort()
        num_ids = [professions_dict[b] for b in a]
        digits = num_ids
        name = {1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9, 10: 0,11:'.'}  # ,"xyz": 3}
        numf = [name[b] for b in digits]
        translation = {39: None}
        # Printing list using translate Method
        dig1 = str(numf).translate(translation)
        dig=dig1.replace(',', '')



        preds =numf
        return preds

@app.route('/', methods=['GET'])
def index():
    # Main page
    return render_template('index.html')


@app.route('/predict', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # Get the file from post request
        f = request.files['file']

        # Save the file to ./uploads
        basepath = os.path.dirname(__file__)
        file_path = os.path.join(
            basepath, 'uploads', secure_filename(f.filename))
        f.save(file_path)
```

```
# Make prediction
preds = model_predict(file_path, model)

# Process your result for human
# pred_class = preds.argmax(axis=-1)            # Simple argmax
   # ImageNet Decode
result = str(preds)               # Convert to string
dig = preds
integers = dig
strings = [str(integer) for integer in integers]
a_string = "".join(strings)
c_r = float(a_string)
mark = 'Current Reading is                      : '
c_r1 = mark + str(c_r)

unit = np.subtract(c_r, pmr)
jack = 'Unit Consumed                           : '
unit1 = jack + str(unit)

price = 24

sani = 'Price per unit                              : '
price1 = sani + str(price)

t_bill = np.multiply(unit, price)
t_bill1 = math.floor(t_bill)
john = 'Total bill is                                 : '
t_bill = john + str(t_bill1)
d_bill = t_bill1 +300
ronal = 'Bill After Due(29June2021)          '
d_bill = ronal + str(d_bill)


class PDF(FPDF):
    def header(self):
        # Logo
        self.image('s.jpg', 10, 8, 20)
        # font
        self.set_font('helvetica', 'B', 20)
        # Padding
        self.cell(80)
        # Title
        self.cell(30, 20, 'Dummy Electricity Bill'    , border=0, ln=1, align='C')
        # Line break
        self.ln(20)

    # Page footer
    def footer(self):
```

```
            # Set position of the footer
            self.set_y(-15)
            # set font
            self.set_font('helvetica', 'I', 8)
            # Page number
            self.cell(0, 10, f'Page {self.page_no()}/{{nb}}', align='C')

    # Create a PDF object
    pdf = PDF('P', 'mm', 'Letter')

    # get total page numbers
    pdf.alias_nb_pages()

    # Set auto page break
    pdf.set_auto_page_break(auto=True, margin=15)

    # Add Page
    pdf.add_page()
    pdf.set_text_color(100,20,400)
    pdf.set_font('helvetica', 'BIU', 14)
    pdf.cell(0, 20, sap, ln=True, border=0)

    pdf.image('meter.jpg', x=-0.5, w=160)
    # specify font
    pdf.set_text_color(50, 50, 50)
    pdf.set_font('times', '', 12)
    pdf.cell(200, 10, '
    Figure 1:Meter Image     ', ln=True, border=0)
    pdf.set_text_color(220, 50, 50)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.cell(0, 10, m_id, ln=True, border=0)
    pdf.set_text_color(100, 50, 50)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.cell(0, 10, p_m_r1, ln=True, border=0)
    pdf.set_text_color(150, 150, 50)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.cell(0, 10, c_r1, ln=True, border=0)
    pdf.set_text_color(80, 80, 80)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.cell(0, 10, unit1, ln=True, border=0)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.set_text_color(10, 80, 100)
    pdf.cell(0, 10, price1, ln=True, border=0)
    pdf.set_font('helvetica', 'BIU', 11)
    pdf.set_text_color(50, 50, 50)
    pdf.cell(0, 10, t_bill, ln=True, border=0)

    pdf.set_font('helvetica', 'BIU', 11)
```

```
        pdf.set_text_color(100, 10, 50)
        pdf.cell(0, 10, d_bill, ln=True, border=0)


        pdf.output('Bill.pdf')

        k = sendpdf("saadrao209@gmail.com",
                    "saadrao209@gmail.com",
                    "03163912619",
                    "Dummy Electricity Bill",
                    "This is your electricity bill.
                    Please pay your bill before 29 June 2021
                    to avoid any discontinuity in the service. ",
                    "Bill",
                    "C:/Artificial Intelligence/testmaskrcnn")

        k.email_send()
        translation = {39: None}
        # Printing list using translate Method
        dig1 = str(result).translate(translation)
        result = dig1.replace(',', '')
        return result
    return None


if __name__ == '__main__':
    app.run(debug=True)
```

Bibliography

[1] Y. Gao, C. Zhao, J. Wang, et al., "Automatic watermeter digit recognition on mobile devices," in Internet Multimedia Computing and Service, 87–95, Springer Singapore (2018).

[2] Y. Kabalci, "A survey on smart metering and smart grid communication," Renewable and Sustainable Energy Reviews 57, 302 – 318 (2016).

[3] M. Vanetti, I. Gallo, and A. Nodari, "Gas meter reading from real world images using a multi-net system," Pattern Recognition Letters 34(5), 519–526 (2013).

[4] D. Quintanilha et al., "Automatic consumption reading on electromechanical meters using HoG and SVM," in Latin American Conference on Networked and Electronic Media, 11–15 (2017).

[5] V. C. P. Edward, "Support vector machine based automatic electric meter reading system," in IEEE International Conference on Computational Intelligence and Computing Research, 1–5 (2013).

[6] Du, M. Ibrahim, M. Shehata, et al., "Automatic license plate recognition (ALPR): A stateof-the-art review," Trans. on Circuits and Systems for Video Technology 23, 311–325 (2013).

[7] D. Karatzas et al., "ICDAR 2015 competition on robust reading," in International Conference on Document Analysis and Recognition (ICDAR), 1156–1160 (2015).

[8] Y. Zhang, S. Yang, X. Su, et al., "Automatic reading of domestic electric meter: an intelligent device based on image processing and ZigBee/Ethernet communication," Journal of RealTime Image Processing 12, 133–143 (2016). Electronic Publication: Digital Object Identifiers (DOIs):

[9] Y.LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature 521(7553), 436 (2015)

[10] Rodriguez, G. Berdugo, D. Jabba, et al., "HD MR: a new algorithm for number recognition in electrical meters," Turkish Journal of Elec. Engineering Comp. Sciences 22, 87–96 (2014).

[11] M. Cerman, G. Shalunts, and D. Albertini, "A mobile recognition system for analog energy meter scanning," in Advances in Visual Computing, 247–256, Springer International Publisher (2016).

[12] V. C. P. Edward, "Support vector machine based automatic electric meter reading system," in IEEE International Conference on Computational Intelligence and Computing Research, 1–5 (2013).

[13] I.Gallo, A. Zamberletti, and L. Noce, "Robust angle invariant GAS meter reading," in International Conference on Digital Image Computing: Techniques and Applications, 1–7 (2015).

[14] M. Majchrak, J. Heinrich, P. Fuchs, and V. Hostyn, "Single phase electricity meter based on mixed-signal processor msp430fe427 with PLC modem," in Proc.17th International Conf. on Radioelektronika, , Apr. 2017.

[15] G. Barbose, C. Goldman, and B. Neenan, "A survey of utility experience with real time pricing," Lawrence Berkeley National Laboratory, Tech. Rep., Dec. 2019.

[16] T. Chandler, "The technology development of automatic metering and monitoring systems," in Proc. IEEE International Power Eng., Dec. 2016.

[17] G. T. Heydt, "Virtual surrounding face geocasting in wireless ad hoc and sensor networks," Electric Power Quality: A Tutorial Introduction, vol. 11, no. 1, pp. 15–19, Jan. 1998. pp. 2214-2217. IEEE, 2007.

[18] L. Gomez, M. Rusi ´ nol, and D. Karatzas, "Cutting sayre's knot: Reading scene text without ˜ segmentation. application to utility meters," in 13th IAPR International Workshop on Document Analysis Systems (DAS), 97–102 (2018)

[19] A. Nodari and I. Gallo, "A multi-neural network approach to image detection and segmentation of gas meter counter.," in IAPR Conference on Machine Vision Applications, 239–242 (2011).

[20] J. C. Gonc¸alves, "Reconhecimento de d´ıgitos em imagens de medidores de consumo de gas natural utilizando t ´ ecnicas de vis ´ ao computacional," Master's thesis, Universidade Tec- ˜ nologica Federal do Paran ´ a - UTFPR (2016).

[21] Shu, Dongmei, Shuhua Ma, and Chunguo Jing. "Study of the automatic reading of watt meter based on image processing technology." In 2007 2nd IEEE Conference on Industrial Electronics and Applications, pp. 2214-2217. IEEE, 2017.

[22] S. Montazzolli and C. R. Jung, "Real-time brazilian license plate detection and recognition using deep convolutional neural networks," in 30th Conference on Graphics, Patterns and Images (SIBGRAPI), 55–62 (2017).

[23] J. Redmon, S. Divvala, R. Girshick, et al., "You only look once: Unified, real-time object detection," in IEEE Conference on Computer Vision and Pattern Recognition, 779–788 (2016).

[24] G.R.Gonc¸alves, M. A. Diniz, R. Laroca, et al., "Real-time automatic license plate recognition through deep multi-task networks," in 31th Conference on Graphics, Patterns and Images (SIBGRAPI), 110–117 (2018).

[25] Ren, Shaoqing, et al. "Faster r -cnn: Towards real -time object detection with region proposal networks." Advances in neural information processing systems. 2015.

[26] B.Shi, X. Bai, and C. Yao, "An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition," IEEE Transactions on Pattern Analysis and Machine Intelligence 39, 2298–2304 (2017).

[27] R. Smith, "An Overview of the Tesseract OCR Engine," Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), 2007.

[28] Guo, Gongde, et al. "KNN model-based approach in classification." OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Springer, Berlin, Heidelberg, 2013.

[29] Dutta, Abhishek, Ankush Gupta, and Andrew Zissermann. "VGG image annotator (VIA)." URL: http://www. robots. ox. ac. uk/ vgg/software/via (2016).

[30] Dutta, Abhishek, Ankush Gupta, and Andrew Zissermann. "VGG image annotator (VIA)." URL: http://www. robots. ox. ac. uk/ vgg/software/via (2016).