Work Package XX : Work Package Title

CADM Number XXXX

Review of algorithms for Machine Learning Approach

Project

Acronym	**eHermes Prototype α**
Title	Annotation of Images for Machine Learning Applications
CADM Number	XXX
Issue Number	001
Authors	Nitha Elizabeth John, Sampreet Sarkar
Contributor	XXX
Reviewers	Benjamin KAWAK
Date	23rd February, 2021

Executive Summary

The aim of this document is to provide a review of the different neural network models, frameworks, and strategies that we have used in our multi-class detection problem, and to be able to reach an unified consensus about which model to select in order to move forward. We have presented the various deep learning frameworks we have considered, and we report our findings. The document is structured according to the following order:

- Section 1 is where we introduce our problem statements and identify our objectives. We seek to clarify the definitions and context of certain aspects of our work, such that we can begin to explain the model architecture and overall process.
- In Section 2, we continue our exploration into the YOLOv3 Neural Network model, and we highlight the various approaches we took to train and infer with our model.
- In Section 3, we explore our approaches on an updated object detector model, the YOLOv4.
- In Section 4, we provide comparisons and evaluate key performance indices between models to aid our conclusion.

Document history

Version	Date	Comments

Applicable documents

AD	CADM	Title

Reference documents

RD	CADM	Title

Contents

Figures

Tables

Acronyms & Abbreviations

Term	Description
Blob	Binary Large Object
COCO	Common Objects in Context
Dnn	Deep Neural Networks
IoU	Intersection over Union
mAP	Mean Average Precision
YOLO	You only Look Once

1 Introduction

In today's world, where the amount of research on Artificial Intelligence and submodules of it grows exponentially, we find ourselves overwhelmed by the volume of Deep Learning frameworks, networks, and algorithms that we can use to achieve our tasks for a particular application. As such, it becomes imperative that we develop a broad understanding of what our objectives are, and by doing so, minimize the search space, pruning frameworks and networks based on key attributes that we care about. Some of the abovementioned attributes could be Model weight, Inference speed, Accuracy of Predictions, and so on. There exist a number of challenges that help us gain a better understanding of how to infer from the metrics of a pre-existing Neural Network architecture, and how to evaluate a custom Neural Network against available solutions. The challenge and subsequently the performance metric that started the renaissance of research into Deep Neural Networks and Convolutional Neural Networks was the ImageNet Large Scale Visual Recognition Challenge[1], which challenged Neural Networks with the then daunting task of object recognition. Since then, we have come a long way in terms of evaluation, and the current trend which has been widely accepted by industry and academia alike has been the Microsoft COCO mAP Metric[2], spearheaded by the MS COCO dataset, containing images spanning over 80 different classes of objects.

Our aim with this document is to provide valuable insight on the object detection algorithm that we found to be most suitable for our task of counting the number of trees in a vineyard, and document the benefits and disadvantages of each model. When we have this documentation at our disposal, we can compare the algorithms and make a choice based on our key performance criteria.

Conventional wisdom instructs us to start designing our Neural Network by using one of the various Deep Learning frameworks available like TensorFlow, Caffe, or PyTorch. The way we would go about with this is we start with an Input Layer which accepts an image in our case[3], and then keep adding Convolutional Layers, and Pooling Layers, and eventually return predictions at the Output Layer. This is often a time-consuming process, but the rewards of doing this are that we get a Neural Network model tailored exactly to our specifications. A better way to go about with is to take a predefined Neural Network model, which has been tested and proven to work with the general aspect of our application, and re-train it with our data of choice to suit our application. It is to be noted that we are only really concerned about the finished product at the end of this training procedure (namely the weights and configurations), and once trained, we can deploy this model on to our application with relative ease and use it for inference.

Before we move on with the analysis of the various models and the approaches that we took, let us take a moment to justify our choice of Neural Networks here. If we go to the basics of TensorFlow, we will see that we find tutorials on how to construct a simple Image Classification tool. While this is a good starting point, it is pretty nascent for our objectives. We would like something on the grounds of an Object detector, preferably a multi object detector. For the sake of producing results fairly quickly and accurately, we would like to choose some kind of a bounding box based detector,

[1] http://www.image-net.org/challenges/LSVRC/index
[2] https://cocodataset.org/#home
[3] The Input Layer generally "flattens" the image which is a Matrix into a one-dimensional vector

instead of a segmentation or mask based detector. A graphical representation of the various types of tasks we can perform through Convolutional Neural Networks is shown in Figure 1 below.

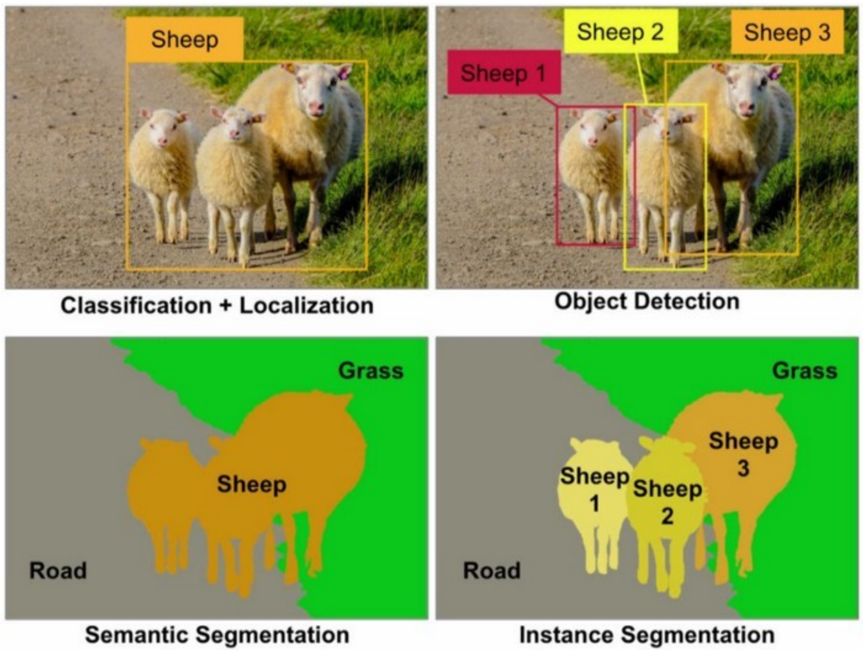

Figure 1: The various tasks performed through Computer Vision and Deep Learning

With all this information at hand, we can finally narrow down our search space into a few top contenders for our application, which are provided as follows:

- **YOLOv3:** YOLO, or You only Look Once is a simple object detector based on top of the Darknet Neural Network. It caused a lot of stir in the academic community when the first model was published by Joseph Redmond in 2018 [1]. It has shown great success in the fields of portability and real-time detection accuracy while still maintaining a relatively simple framework and ease of operation.

- **YOLOv4:** In April of 2020, Alexey Bochkovskiy published a paper with a significant update to the existing YOLOv3, entitled "YOLOv4: Optimal Speed and Accuracy of Object Detection" [2]. In this paper, the authors proposed a much more robust network, which had added functionalities and more layer types, along with significant changes to the darknet framework behind the detector as well.

2 Analysis of the YOLOv3 model

There exists a lot of good object detection neural networks, which provide a very high accuracy, however the approach is purely for research purposes, as the time taken to make a prediction is quite high. As a result, these Neural Networks could not be implemented for practical applications, and more importantly for real-world applications *(like smartphone apps or a live camera feed)*. This is the problem YOLOv3 sets out to solve, with

2.1 Approach 1: Using Darknet for Training

In this approach, the idea was to use the darknet framework to train the YOLOv3 model, and use the model for inference as an OpenCV application. This approach is by far the most well-documented one, since this is the way the YOLO detector was conceived to be used. The main idea here is to use the `darknet` tool to train our customized model on the dataset, which would generate a weight file, containing the weights of the neurons in the network. These weights along with the configuration file, can be used to recreate the model at run-time, for inference purposes. We will highlight the process which we followed to build and utilise the darknet tool for training our YOLOv3 model.

2.1.1 Steps to install the darknet tool

Installing the darknet library is quite simple. We need to fetch the sources from GitHub and compile it. These steps are highlighted below:

```
$: git clone
$: cd darknet && make
```

This would install darknet with the "default" settings, *i.e.,* with no GPU support, and without OpenCV support. However, if you want to install the library with GPU and OpenCV support, in that case, you can open the Makefile, make the following changes, save and rerun the make command. The changes are highlighted as follows:

- You can open the Makefile by using the command `nano Makefile`
- You need to edit the first and second lines to reflect the fact that you have a CUDA compatible GPU. Set `GPU=1`, `CUDNN=1`, and `OPENCV=1` in the Makefile.
- Save and exit from the Makefile
- Re-run the compile process by using the command `make`.

This will generate the executable called "`darknet.exe`" *(or darknet, on Linux)*. We shall use this executable to run the training process, and we can also use this executable to run inference on the fly. This process should not take long, but it depends on the options set in the Makefile, and the processing speed of the computer.

2.1.2 Creating the dataset for training

We use the labelImg[4] tool to annotate[i,5] our images. For our dataset, we have chosen 1000 images *(800x600 RGB JPEG images)*[ii], and proceeded to annotate them in the YOLOv3 acceptable TXT format. Then we ran a simple Python script[6] which created a text file for us called "`train.txt`", which contains the location of all our training images with respect to the root directory, where the darknet executable resides. We then need to provide a list of classes *(which should be automatically generated while annotating with labelImg)*, along with the train.txt file. The annotation process is quite time consuming, because there is a bit of a learning curve in the beginning, but once you get habituated with the user interface and the keyboard shortcuts, you start to gain momentum. The annotation process took around 12 hours for us to complete for 1000 images.

[4] LabelImg: https://github.com/tzutalin/labelImg

[5] In terms of Deep Learning and Computer Vision, annotation is the process of assigning labels to certain areas of interest in the image to create a labelled dataset.

[6] The Python script ''extract-images.py'' can be found here: treecounter-ML/extract-images.py at main · software3daerospace/treecounter-ML (github.com)

2.1.3 Training the Model on our custom dataset

Now that we have all the files we need, we can use the darknet executable to train our model on our custom dataset. For this purpose, we have already labelled our dataset and generated the train.txt file. The next step is to create a data file, which will essentially point towards all the data and locations to save the models to. For us, the data file looked like the following:

```
classes=3

train=data/dataset/train.txt

names=data/dataset/classes.names

backup=backup/
```

This signifies that we wish to train for 3 classes, namely – *tree*, *wooden post*, and *metal post*, we have our training data at the above-mentioned location, the names file is there to perform a correlation between the names and the numerical classes assigned in the YOLO annotations. We also specify that we want to periodically save the models in the backups directory. We can navigate on to the darknet root directory, and use the following command to start the training procedure:

```
./darknet.exe detector train <path-to-model-cfg-file> <path-to-data-file>
<pretrained-weights>
```

This will start the training procedure, which will open up a chart showing the total loss per iteration. The aim is to minimize the loss function, but keeping in mind to not overfit the model to the data. The training took around 9 hours, at the end of which we had a total model loss[7,iii] of 1.54%. We can see the gradual descent of the loss function[8] in the following Figure 1:

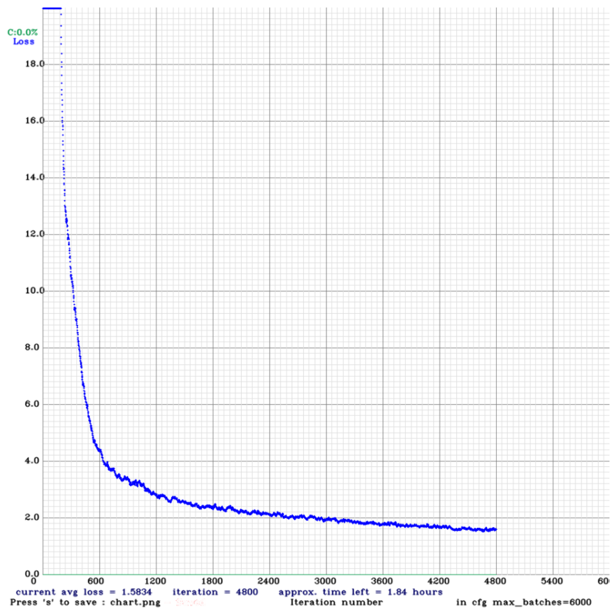

Figure 2: The descent of the loss function

[7] The model loss is an important metric in Neural Network training which distils all aspects of the huge and complex model down to one score, based on which the performance of the Network may be ranked or compared against others. From the perspective of an optimization problem, the loss function can be seen as a minimizing function or a cost function.

[8] The darknet framework uses a default loss function called *IoU: Intersection over Union*

2.1.4 Using the model for inference

Now that our model has trained, and has produced a corresponding weights file, we can use this file in conjunction with the model configuration file for inference purposes. The dnn[iv] module of OpenCV presents us with a lot of options to load weights from various deep learning frameworks, like Darknet, Caffe, TensorFlow, and so on. In our case, we can describe a model by its cfg file and its weights file, which we shall have to pass as parameters in the function `cv2.dnn.readNetFromDarknet(config, weights)` in our main application. Then we need to convert the input image into a blob of size 416x416, and set this blob[v] to be our input layer for the model. From there on, we can generate predictions, an example of which is shown in Figure 2 below.

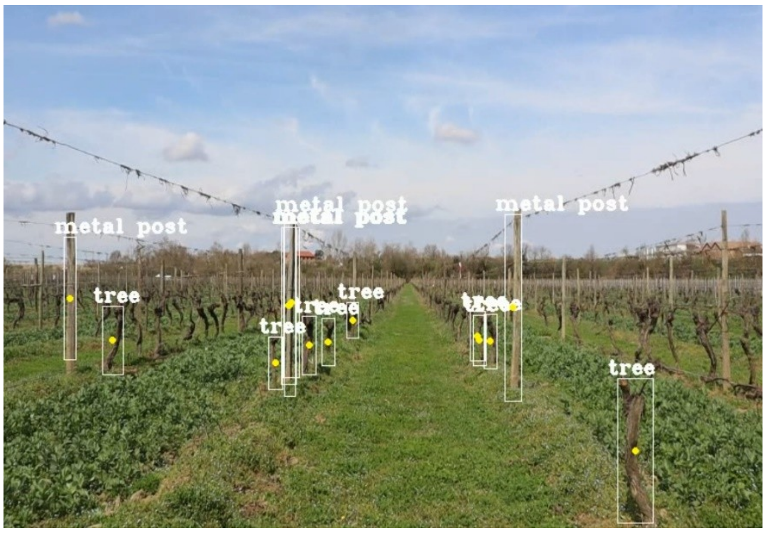

Figure 3: The predictions shown by the bounding boxes

2.2 Approach 2: Using Darknet for Training and TensorFlow for Inference

In this approach, we try to break away from using the OpenCV dependencies and instead gradually move in towards using a standard and highly coveted library used in modern Deep Learning applications – TensorFlow. We do not break away from the Darknet ecosystem entirely, we still use the darknet tool to train our model, however this time, we convert our Darknet weights file into the TensorFlow compatible format – a protobuf file (.pbtxt) and a checkpoint (.ckpt) file, which we can again load in our OpenCV application using the `cv2.dnn.readNetFromTensorFlow(frozen_graph.ckpt, graph.pbtxt)` function.

The one main advantage of converting these weights into TensorFlow is that now we know our weights are a little more optimized that the raw YOLO weights, and these files are acceptable in a wider range of applications. There is also a way to condense the huge size of this protobuf file *(around 250 MB)* into a TFLite format *(10-20 MB)*. It is to be noted however, that using the TFLite

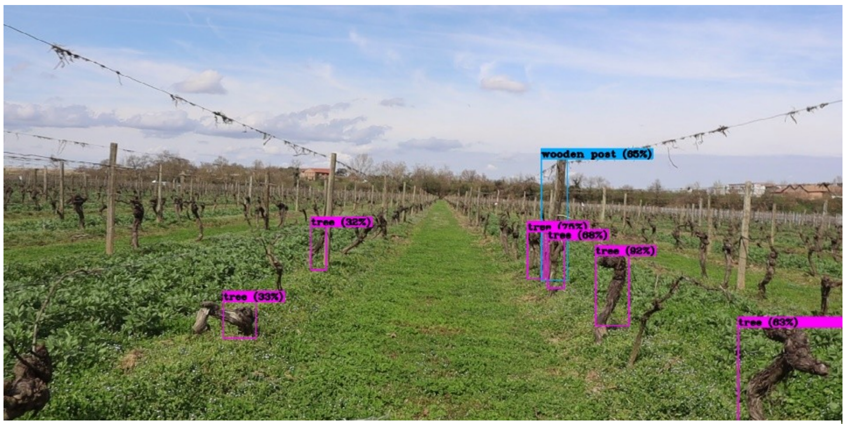

Figure 4: Predictions on TensorFlow model

model for inference might report a slight drop in the accuracy, the trade-off being between model size and accuracy. The predictions returned from this TensorFlow model is shown in Figure 3.

From what we observe, there is not much improvement in the performance. The mean accuracy is around 70%, however the model is still confused between wooden posts and metal posts. This might be fixed by retraining.

3 The YOLOv4 model (on Darknet)

1. <u>Analysis of the YOLOv4 model</u>

YOLOv4 is a one-stage object detection model. The YoloV4 backbone architecture is composed of three parts: Bag of freebies, Bag of specials, CSPDarknet53. YOLOv4 utilizes the CSP connections with the Darknet-53 as the backbone in feature extraction. The CSPDarknet53 model has higher accuracy in object detection compared with ResNet based designs even they have a better classification performance. To enrich the information that feeds into the head, neighboring feature maps coming from the bottom-up stream and the top-down stream are added together elementwise or concatenated before feeding into the head. Therefore, the head's input will contain spatial rich information from the bottom-up stream and the semantic rich information from the top-down stream. This part of the system is called a neck. In YOLOv4, a modified SAM is used without applying the maximum and average pooling. In SAM, maximum pool and average pool are applied separately to input feature maps to create two sets of feature maps. The results are fed into a convolution layer followed by a sigmoid function to create spatial attention. This spatial attention mask is applied to the input feature to output the refined feature maps.

2. <u>Advancement in YOLOv4 in comparison to prior YOLO models</u>

- It is a proficient and authoritative object detection model that allows individuals with a 1080 Ti or 2080 Ti GPU to train a very fast and accurate object detector.

- The consequences of state-of-the-art "Bag-of-Freebies" and "Bag-of-Specials" object detection procedures all the while detector training was confirmed in version 4.

- The converted state-of-the-art methods covering CBN (Cross-iteration batch normalization), PAN (Path aggregation network), that are greater skilled and applicable for single GPU training.

3. <u>Steps Performed</u>

a. Dataset Preparation

Ideally, as in any ML applications, the annotations were carried on our custom dataset for 3 classes (tree, wooden post, metal post). These annotations were obtained in YOLO format in a text file. To avoid over-fitting and achieve an objective evaluation regarding our model, The total of the database must be split into two for the training set and validation set. From the prepared database 30% is used for validation set and rest 70% is used for training set.

b. Configuring Training Pipeline

The major steps involved in setting up a pipeline for training is to modify the network and to create the configuration files. Here, YOLO is created on CSPDarknet53, which is an open source neural network framework to train the detector and it is composed of 53 layers of darknet. The filters of Darknet are by default set to train a network of 80 classes. Hence the number of filters must be modified for the purpose. Along with configuration files two more files have to be refined, which are the .names and .data files. The names file contains the label given (tree, metal post and wooden post) to the images during the annotation step and data files must be scripted the let know the framework the no of classes, data for validation and training

c. Training the model

The process of training neural networks is the most challenging part of using the technique in general and is by far the most time consuming, both in terms of effort required to configure the process and computational complexity required to execute the process. The training is done to learn a mapping approach from the input to output. This is achieved by updating the weights of the network in response to the errors the model makes on the training dataset. Updates are made to continually reduce this error until either a good enough model is found or the learning process gets stuck and stops.

For the experiment carried out for this trial model, Google COLAB was used for performing the training processes for developing the model. It took around 12 hours to complete the training for 6000 batches.

d. Exporting Weights File

The weights obtained will be in Darknet format. The first three int32 values are header information which includes major version number, minor version number, and subversion number, followed by int64 value that is the number of images seen by the network during training. After that, there are 62 001 757 oat32 values which are weights of each convolutional and batch norm layer. It is important to remember that they are saved in row-major format, which is opposite to the format used by the TensorFlow model. Now these weights have to be exported and stored for future usage.

e. Testing of the model

The detection was implemented with the help of YOLO weights that have been obtained in the previous step. That is the testing is carried out using the weights obtained in Darknet format.

4. <u>Result and issues faced</u>

It was observed that the model resulted in a mAP of 70% (overall average accuracy for all 3 classes). The tree category alone had an accuracy nearly 85%, however the category wooden post resulted in accuracy of 48% and metal post 75%. Hence the overall accuracy can be improved by improving the dataset by adding more images of wooden posts and metal posts. 1.

The metal post is not always detected as METAL posts, even human programmer fails in identifying whether its metal or wood. This can also be refined by adding more images of the post and also may be by removing the category of "wooden post".

The training time was nearly 12 hours and this can be improved by using a system with better GPU configurations.

5. <u>Comparison of YOLOv4 with other models</u>

During this experiment, YOLOv4 achieved a mAP value of 69.5% over the custom dataset created, and gained a real-time speed of almost 17 FPS on the google COLAB, outperforming the swift and highly accurate detectors in the particulars of both speed and accuracy.

YOLOv4 is double as rapid as EfficientDet beside corresponding efficiency, also in comparison to YOLOv3, the mAP and FPS have been enhanced by 10% and 12%, respectively.

4 The YOLOv4 model and inference on TensorFlow

(To be done)

5 Comparisons and Key Performance Index:

5.1 Key Performance Index

Criteria	YOLOv3	YOLOv4
Annotation Time	12 hours	12 hours
Training Time	9 hours	10 hours

Detection Time	9 FPS	16 FPS
Model Size	234 MB	250 MB
Accuracy	64.1%	69.5%

Table 1: Comparisons based on Key Performance Index of the two algorithms

6 References

[1] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018.

[2] A. Bochkoviskiy, C. Y. Wang and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934.,* 2020.

[i] A detailed review on the image annotation procedure can be found in the Reference Document titled "Annotating Images for Machine Learning Purposes"

[ii] The choice of the image format can be accredited to the following reasons:

- **Space-saving:** Since there is a limited amount of onboard storage capacity in the eHermes receiver, we have to be careful with how much space each image takes up in the system. The JPEG compression, albeit lossy, provides a good amount of detail to work with, while not taking up a huge amount of space.
- **OG02B10 Camera capabilities:** Since we are eventually going to use the OG02B10 cameras on the actual hardware, we have to be considerate of its capabilities as well. The camera can output in three native resolutions: 640x480, 1280x720, and 1600x1300. Since 640x480 would be too small to be able to extract precise features, we chose to perform these reviews with the 800x600 resolution, which we can downsample from 1280x720 images without a significant loss of quality.

[iii] The darknet framework provides a choice of loss functions to choose from, out of which the default loss function setting is the IoU (Intersection over Union), or rather the Generalised IoU. The IoU is particularly interesting in terms of object detection Neural networks, because it provides a metric based on the area of overlap between the actual labelled bounding box, and the predicted bounding box. Why this is of interest to us, is primarily because we can have a good idea of how accurately our model can predict the given classes of objects by looking at the IoU value. We would love an IoU value of 1.0, which would go on to signify that our actual bounding box and the predicted bounding box are in perfect overlap, however this is seldom the case in the real world. A good IoU value is anything greater than 0.75, or a 75% overlap. This is because we have to leave room for the different aspect ratio of images.

[iv] The Deep Neural Network module provided by the Open-Source Computer Vision Library, or OpenCV in short, has been a game changing event in the field of merging Computer Vision with Artificial Intelligence, Deep Learning, to be precise. The module is available in the core C++ library, as well as the more popular Python wrapper *(opencv_python)*, which has added to the benefits. The dnn module allows the programmer to design applications that can seamlessly load trained models from various Deep Learning frameworks, such as Caffe, TensorFlow, Darknet, etc. The methods in this module allow us to load a model from disk, and use the model for inference at run-time, thus increasing the productivity and separating the Computer Vision application from the Deep Learning model. In our case, we have been able to repurpose the same application code for two vastly different frameworks by essentially changing two lines of code in our application.

[v] A Blob, or Binary Large Object, and is used to define a group of connected pixels in a binary image. It is seldom the case that an object of interest to us (which belongs to the real world), will have a perfect shape, which can be defined by very simple mathematical relations. Often, we find that our objects of interest are of various shapes and form, hence it becomes much simpler if we look for the connections within the pixels to estimate a relation rather than looking at a simple shape-based relation. For example, if the object of interest is a brick, then as the camera moves around the object, we find the same brick assuming different forms, depending upon the portion of the total surface area that is visible to the camera. Hence, trying to detect this brick using predefined notions about its shape would not return the best results. We can try to estimate the shape by looking at (say), the luminance of the pixels and the relation it has with their immediate neighbours. Since Deep Neural Networks rely heavily on being independent of scaling and orientation of the object of interest, it suits our needs better to imagine and process sections of our image as blobs, rather than the whole image itself.