



**Work Package XX : Work Package Title**

**CADM Number XXXX**

**Review of algorithms for Machine Learning Approach**

**Project**

|  |  |
| --- | --- |
| **Acronym** | **eHermes Prototype α** |
| **Title**  **CADM Number**  **Issue Number** | Annotation of Images for Machine Learning Applications  XXX  001 |
| **Authors**  **Contributor**  **Reviewers** | Nitha Elizabeth John, Sampreet Sarkar  XXX  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

[Executive Summary 3](#_Toc66468284)

[1 Introduction 6](#_Toc66468285)

[2 Analysis of the YOLOv3 model 6](#_Toc66468286)

[2.1 Approach 1: Using Darknet for Training 6](#_Toc66468287)

[2.1.1 Steps to install the darknet tool 6](#_Toc66468288)

[2.1.2 Creating the dataset for training 6](#_Toc66468289)

[2.1.3 Training the Model on our custom dataset 7](#_Toc66468290)

[2.1.4 Using the model for inference 7](#_Toc66468291)

[2.2 Approach 2: Using Darknet for Training and TensorFlow for Inference 8](#_Toc66468292)

[2.3 Approach 3: Using TensorFlow for training and Inference 9](#_Toc66468293)

[3 Approach 3: The YOLOv4 model (on Darknet) 9](#_Toc66468294)

[4 Approach 4: The YOLOv4 model and inference on TensorFlow 11](#_Toc66468295)

[5 Comparisons and Key Performance Index: 11](#_Toc66468296)

[5.1 Key Performance Index 11](#_Toc66468297)

Figures

[Figure 1: The descent of the loss function 7](#_Toc66468298)

[Figure 2: The predictions shown by the bounding boxes 8](#_Toc66468299)

[Figure 3: Predictions on TensorFlow model 8](file:///C:\Users\visio\Documents\Important%20Documents\3D%20Work%20Template.docx#_Toc66468300)

Tables

Table 1: Comparisons based on Key Performance Index of the two algorithms …………………………….. 12

Acronyms & Abbreviations

|  |  |
| --- | --- |
| **Term** | **Description** |
|  |  |
|  |  |
|  |  |

# Introduction

# Analysis of the YOLOv3 model

For the sake of going chronologically, we shall start with the YOLOv3 model.

## Approach 1: Using Darknet for Training

In this approach, the idea was to use the darknet framework to train our 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.

### 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 <https://www.github.com/pjreddie/darknet.git>  
 $: 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.

### Creating the dataset for training

We use the labelImg tool to annotate our images. For our dataset, we have chosen 1000 images*(800x600 RGB JPEG images)*, and proceeded to annotate them in the YOLOv3 acceptable TXT format. Then we ran a simple Python script 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.

### 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, 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 of 1.54%. We can see the gradual descent of the loss function in the following Figure:

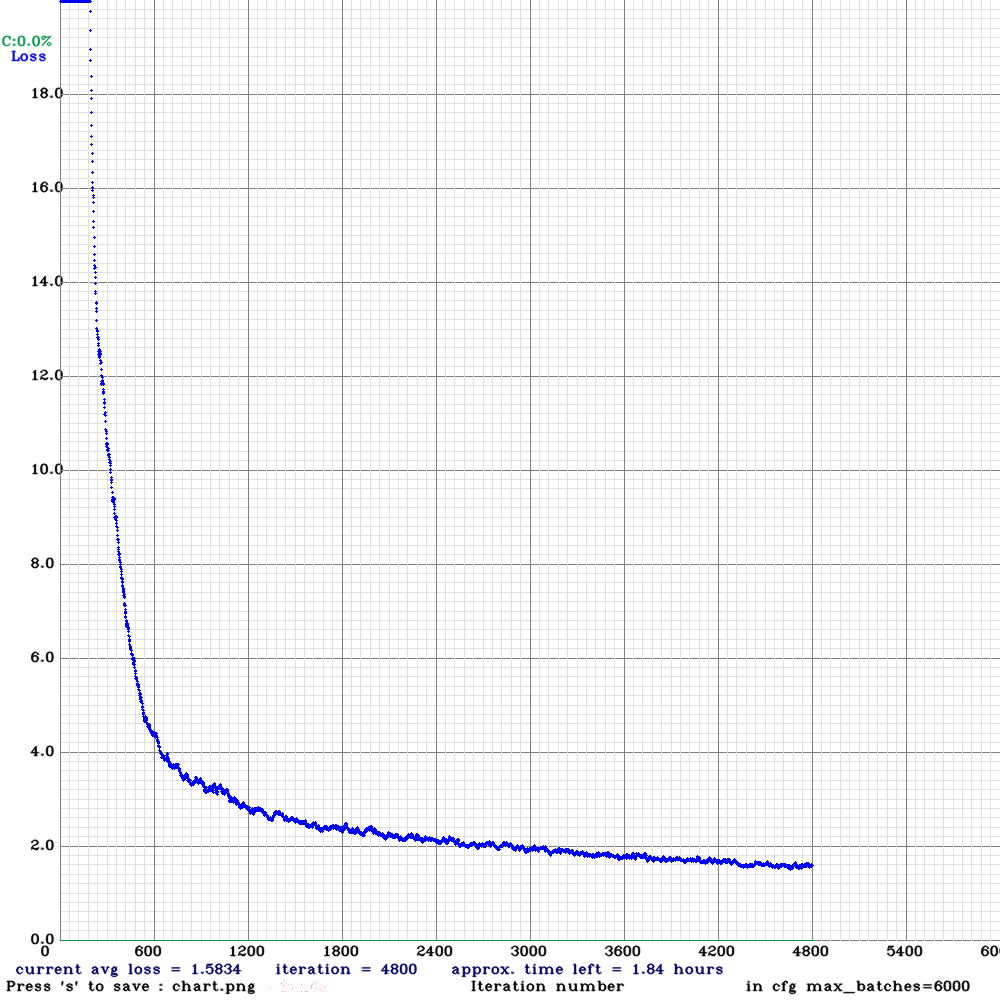


Figure : The descent of the loss function

### 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 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 to be our input layer for the model. From there on, we can generate predictions, an example of which is shown below.



Figure : The predictions shown by the bounding boxes

## 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 (.pb) and a protobuf text file, which we can again load in our OpenCV application using the cv2.dnn.readNetFromTensorFlow(frozen\_graph.pb, 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 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.

Figure : Predictions on TensorFlow model

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.

## Approach 3: Using TensorFlow for training and Inference

(To be done)

# Approach 3: The YOLOv4 model (on Darknet)

**1.** **Analysis of the YOLOv4 model**

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.** **Advancement in YOLOv4 in comparison to prior YOLO models**

• 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.** **Steps Performed**

**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 techniquein general and is by far the most time consuming, both in terms of effort required toconfigure the process and computational complexity required to execute the process. Thetraining is done to learn a mapping approach from the input to output. This is achievedby updating the weights of the network in response to the errors the model makes on thetraining dataset. Updates are made to continually reduce this error until either a goodenough 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 performingthe 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 headerinformation which includes major version number, minor version number, and subversionnumber, followed by int64 value that is the number of images seen by the network duringtraining. After that, there are 62 001 757 oat32 values which are weights of eachconvolutional and batch norm layer. It is important to remember that they are saved inrow-major format, which is opposite to the format used by the TensorFlow model. Now theseweights 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 obtainedin the previous step. That is the testing is carried out using the weights obtained in Darknet format.

**4.** **Result and issues faced**

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.** **Comparison of YOLOv4 with other models**

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.

# Approach 4: The YOLOv4 model and inference on TensorFlow

(To be done)

# Comparisons and Key Performance Index:

## 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